DEPARTMENT OF COMPUTER SCIENCE VI
AUTONOMOUS INTELLIGENT SYSTEMS

UNIVERSITY OF BONN

Master Thesis

Continuous Motion Planning for Domestic Service Robots with Multiresolution in Time

Ricarda Steffens

Primary reviewer:	Prof. Dr. Sven Behnke
Secondary reviewer:	Prof. Dr. Rolf Klein
Supervisor:	Dipl.-Inform. Matthias Nieuwenhuisen

December 2, 2013

Declaration of Authorship

I hereby declare that I have prepared this thesis independently and without the use of sources and aids other than those stated. All sources have been cited accordingly.

Bonn, December 2, 2013

Ricarda Steffens

Abstract

In this thesis I present an approach to continuous motion planning with multiresolution in time. The approach is based on the motion planning algorithm STOMP by Kalakrishnan et al. and designed to decrease the overall runtime of the efficient optimization algorithm in order to enable frequent replanning. Since domestic service robots operate in household environments with dynamic obstacles, it is likely that planned trajectories become invalid over time. Thus, it is not necessary to provide trajectories with a high resolution throughout time. The multiresolutional approach implicitly considers the uncertainty of the future by providing a trajectory with a gradually coarser schedule, which is refined while replanning. Besides employing the multiresolution in time, the fact, that the STOMP algorithm converges faster with a good guess for an initial trajectory, is utilized to speed up the optimization process. As a result of the decreased runtime of the algorithm, joints additional to the robot arm can be included into the motion planning. The improved implementation of STOMP is evaluated in simulation with regard to runtime and success in avoiding static and dynamic obstacles. It is directly compared to the original STOMP implementation as well as to a kinodynamic motion planning algorithm called KPIECE. By means of the conducted experiments I show that the improved STOMP implementation is superior to both motion planners concerning the overall runtime and moreover, is able to successfully avoid dynamic obstacles.

Contents

1 Introduction

As the population in first world countries grows older and in general people have less time to do chores, domestic service robots become more and more important and will also gain popularity. In contrast to working in industrial mass production, domestic service robots have to operate in household environments, which are especially designed to comply to humans. Therefore, the robots have to adapt to their surroundings by possessing a suitable body composition as well as certain skills. They require capabilities such as the perception of their environment, robust navigation and mobile manipulation. All of these skills accompany each other when solving varying tasks, for instance picking up objects.

Because of the robots directly working next to the human users, it is important that safety requirements are met. Therefore, regardless of the specific task the robot is executing, it has to avoid collisions as soon as it navigates or moves other body parts. It is crucial to prevent the user from being harmed and similarly, it should be assured that neither the robot nor furniture gets damaged. Hence, obstacles have to be considered while planning the movement of the robot. Not only do the robots have to face household environments crowded with static obstacles like chairs and bottles, but they also have to cope with a highly dynamic world. As a result, the robot has to rapidly adapt to occurring and shifting obstacles, such as people, pets or objects being utilized, in order to avoid them. Of course, self-collisions of the robot with its static and moving body parts have to be also prevented.

In spite of motion planning being computationally expensive, it is necessary to employ fast motion planning with constant replanning in order to react without much delay. Apart from this, the planned motions should be feasible and not jerky, because otherwise, the movement of the robot becomes non-predictable for human users and violations of joint limits may occur.

Although the robot is able to avoid numerous collisions by employing motion planning, finding a collision-free path is not for certain. It is possible to increase the probability of finding a feasible movement by including not only the robot arm but also additional joints from the upper body of the robot into the planning process. Unfortunately, including more joints leads to a higher computational complexity and therefore, slows down the whole motion planning process. Hence, a fast motion planning implementation becomes even more crucial in order to provide successful planning.

To date, our domestic service robots Dynamaid and Cosero (cf. Fig. 1.1) are able to prevent collisions during navigation by means of steering their bases. In addition, arm motion planning is employed when grasping objects out of a box [10]. Still, no torso joint is utilized for the motion planning and due to the lack of replanning for the arm, dynamic obstacles cannot be avoided. Moreover, planning only takes place when executing a bin picking task while otherwise, predefined motion primitives are used for the motions of the arm, which sometimes leads to an artificial shape of movements. Overall, the current status of our robots necessitates the implementation of superior motion planning in order to enhance their motion and collision avoidance abilities.

So far, our robots use a customized version of the motion planning pipeline provided

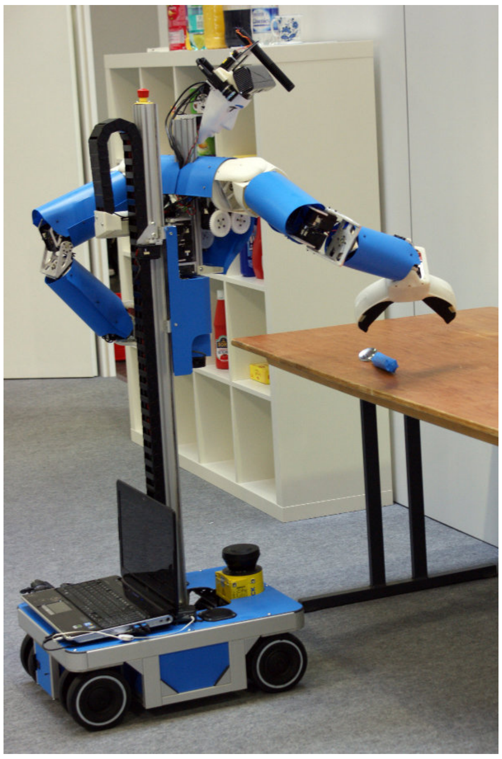


Figure 1.1: Domestic service robot Cosero executing a task in a household environment.

by the middleware ROS by Willow Garage [12]. It includes the Open Motion Planning Library (OMPL) [19] and utilizes Kinodynamic Planning by Interior-Exterior Cell Exploration (KPIECE) by Şucan and Kavraki [18] as the algorithm to find a feasible plan. Although there are various optimization algorithms for motion planning, the Stochastic Trajectory Optimization for Motion Planning (STOMP) by Kalakrishnan et al. [8] algorithm is promising. It has a lot of advantages such as being able to utilize a non-differentiable cost function and not needing further trajectory filtering. Despite its disadvantages like converging slightly slower compared to for instance gradient-based optimization methods because of its stochastic approach, it is reasonable to use STOMP as a starting point for employing fast and efficient motion planning. This can be realized by using different techniques for instance a multiresolutional approach.

Subsequently, I will present the problem, which specifies the topic of this master thesis, in more detail. This is followed by an overview of the approach I will apply in order to solve the described problem.

1.1 Problem specification

When utilizing motion planning for a robot, the goal is to compute feasible and collision-free trajectories between a start and a goal configuration. Furthermore, the best solution out of all feasible solutions should be chosen on account of efficiency. To date algorithms for computing paths to navigate our robots and to plan trajectories for one arm are implemented. However, while the algorithms for navigation are fast, the motion planning for the endeffector is very slow due to the curse of dimensionality. Therefore, the motion planning is rarely employed and instead, predefined motion primitives are preferred.

In order to take obstacles and possible collisions into account, a model of the robot and the environment is required. For this reason the sizes and positions of every robot part in the robot's body composition have to be determined and translated into a representation of the robot. Also, the sensor data obtained by different sensors have to be processed in order to generate an environment model. By means of the robot and environment representations, implementations of given collision detection algorithms can be utilized to detect collisions with obstacles as well as self-collisions. Since a huge amount of calculations have to be executed, the characteristics of both representations have to be chosen wisely. On the one hand they should not be too detailed as this would lead to increased computations, but on the other hand the models have to be adequate in order to not miss collisions. So far the specified robot and environment models of the ROS planning pipeline are utilized, when the motion planning is running. Still, both representations could be refined further.

As household environments commonly not only include static obstacles, but also dynamic ones, it is crucial to consider both of them. Due to the motion of those dynamic obstacles as well as the movement velocities of the robot itself, the planning process needs to be fast and similarly, frequent replanning is required to rapidly adapt to changes in the environment. As our robots mostly move their arms by means of motion primitives, no collision avoidance takes place whatsoever. This yields to safety issues, since collisions with obstacles and even human users can occur. Apart from this, the present motion planning software only plans once for each call and subsequently executes the planned trajectory without renewed collision checking. Unfortunately, even with constant replanning the motion planning process might be too slow to react accordingly.

Usually, motion planning is limited to navigation as well as arm motion planning to move the end-effector of a robot. After all, robots consist of different body parts with many additional joints, which also should not be disregarded. Those additional joints can assist with finding superior or actually collision-free trajectories, when the motion planning for the arm did not succeed in the first place. Although many different possibilities for avoiding collisions are gained, the extra joints cause a huge amount of calculations concerning collision detection, inverse and forward kinematics and the optimization process itself. To date our robots only plan paths for the arm, while their torso joints are not included into planning, in spite of the torso joints and especially the torso elevator enabling numerous options for movements.

Overall, the resources for computation are limited because of the employed laptop hardware. Accordingly, all parts of the motion planning process and particularly those which rely on heavy computation, have to be improved. Hence, it is again very important to speed up the whole motion planning process to allow constant replanning as well as incorporating additional joints.

In addition to available capabilities, safety is an important aspect for domestic service robots. Consequently, swift and jerky movements of the robot should be avoided. Besides leading to a more smooth and therefore natural and human-like behavior, less force is needed to actuate the robot. While the former leads to movements being predictable for human users, the latter maintains that the robot hardware is conserved. Altogether, smooth trajectories make for efficient motions regarding trajectory length and therefore, execution time.

1.2 Approach

Instead of starting a new motion planner from scratch, it is expedient to use a given planner as an initial point for enhancement. As there are various optimization algorithms for motion planning, it is fundamental to choose a planner, which not only has a promising and working concept, but also is designed in a way that enables various working points for modification. One motion planning algorithm, which lives up to this criteria, is STOMP [8] and consequently, it is chosen as a foundation for this thesis. In Chapter 4 the STOMP algorithm, its implementation and its pros and cons will be described in detail. In addition, STOMP can be easily integrated into the ROS middleware, which is used as a framework for the software of our robots.

In order to tackle the previously defined problems, STOMP provides various working points. On the one hand the configurations of STOMP as well as the input can be improved and on the other hand the planner itself can be changed for enhancement. Overall, it is expedient to simplify representations and calculations and also to limit the given problem by defining constraints and making assumptions.

Regarding the configurations a lot of adjustments can be made, for instance the size and resolution of the environment representation, the number of iterations of the algorithm, the sampling noise or weights for the cost function can be calibrated. Furthermore, new weights and configuration items can be added to tune the performance of the planner. Apart from this, the quality of the planned trajectories can be enhanced by refining the cost function of the planning algorithm in terms of adding new costs constraints.

Additional joints slow down the whole planning process by adding another dimension to the planning space, nevertheless the runtime can be decreased by means of reusing the previously planned trajectory as an input to the planner when replanning. Consequently, the planner will converge faster, particularly with obstacles present. Also collision checking as well as self-collision detection can be fastened up by dints of configuration files defining correlations between the robot's joints and restricting collision checking to the actuated body parts. Thus, the amounts of calculations added by the extra joints regarding collision checking can be limited.

One significant idea of this thesis is to consider the uncertainty of the future, which exists due to a continuously changing environment. Because dynamic obstacles might interfere with the planned trajectory during its execution, it is not reasonable to plan accurate waypoint positions throughout the whole trajectory. Hence, a multiresolutional approach can be employed to implicitly take the uncertainty into account. This leads to the trajectory spacing being more detailed for the beginning of the trajectory and gradually evolving into a rough schedule. Since frequent replanning will be employed, the rough plan becomes more refined after each planning cycle. As a result of the multiresolutional waypoint spacing, the number of waypoints is reduced and for this reason, the optimization process will speed up. Moreover, the quantity of waypoints can be reduced further by initially choosing a reasonable overall duration of the trajectory considering the start and goal configurations. Altogether, the multiresolutional approach will decrease the runtime of the algorithm while maintaining the quality of the computed trajectories.

1.3 Structure of thesis

In the next chapter I will discuss the prior work related to the topic of motion planning and multiresolutional approaches. Then I will outline the fundamentals of motion planning and explicitly describe the given robot hardware and software framework. In the following chapter the STOMP algorithm as well as its implementation will be described. Subsequently, I am going to introduce the approach for solving the demonstrated motion planning problem on which the work of my master thesis focuses. Finally, the methods and experiments employed for evaluation as well as the results will be presented, followed by a summary of my thesis and possible future work.

2 Related Work

In the field of robotics a lot of research has already been conducted regarding motion planning, as it is an essential ability required by every type of robot. In terms of the domain of domestic service robots motion planning focuses on mobile manipulation which includes simple navigation as well as manipulation by means of an endeffector. Due to the composition of robot arms being quite complex, it is not trivial to find feasible and collision-free trajectories fast. Therefore, numerous work engages in arm motion planning.

The research platform PR2 by Willow Garage employs a whole planning pipeline integrated into their middleware ROS [3]. It consists of tools for sensing and modeling the environment, formats for defining a robot model, different sampling-based motion planning algorithms and tools for finally executing the planned motions. The motion planning algorithms are taken from the Open Motion Planning Library (OMPL) by Şucan et al. [19], which for instance includes Rapidly exploring Random Trees (RRT), Probabilistic RoadMaps (PRM) and Kinematic Planning by Interior-Exterior Cell Exploration (KPIECE). All of these algorithms can be utilized and more or less easily interchanged within the planning pipeline. Moreover, it is possible to adapt the components to other robot architectures than the PR2 or just employ parts of the planning pipeline for an own motion planner. Overall, ROS provides a good and working foundation for motion planning tasks. However, just the basic functionality is given and therefore, it is inevitable to modify and improve the components in order to enable features such as frequent replanning.

So far the robots Dynamaid and Cosero from the Autonomous Intelligent Systems Group of the University of Bonn also utilize the ROS planning pipeline for motion planning and in particular, bin picking tasks [10]. After generating a feasible grasp for one out of a pile of objects in a bin, motion planning is employed for moving the endeffector over the box and subsequently, to a pre-grasp position. On that account, motion planning is executed by means of Lazy Bi-directional KPIECE [18], which is included in OMPL. Although the ROS planning pipeline is customized for the robots, the planning is only made once for each object and then the trajectory is executed without further replanning. Also, the whole planning process as well as the execution of the planned trajectory is overall very slow compared to other approaches.

In addition to the sampling-based motion planning algorithms in OMPL, another algorithm is provided in ROS. The Covariant Hamiltonian Optimization and Motion Planning (CHOMP) is a gradient optimization algorithm presented by Ratliff et al. [13]. It uses trajectory samples and performs a covariant gradient descent to find a smooth and collision free trajectory. Unlike other sampling-based planners, the trajectories computed by CHOMP are already smooth and short. Moreover, the initial trajectory does not have to be collision-free in order to lead to a valid plan. When optimizing, the initial trajectory is updated iteratively in terms of minimizing the costs for the trajectory. In order to determine the costs, CHOMP needs a differentiable cost function, which results in a restriction of the planner and the constraints applied to the planned trajectories. CHOMP is able to generate feasible and collision-free trajectories rather fast.

A planning algorithm based on CHOMP is the Stochastic Trajectory Optimization for Motion Planning (STOMP) by Kalakrishnan et al. [8]. STOMP utilizes the advantages of CHOMP combined with a stochastic approach. In contrast to CHOMP it is no longer required to use cost functions for which gradients are available, while the performance of STOMP stays comparable. As I use STOMP as an initial point for the work of this thesis, I will describe the planner in detail in Chapter 4.

Another algorithm lightly based on CHOMP as well as STOMP is ITOMP, an incremental trajectory optimization algorithm for real-time replanning in dynamic environments [11]. In order to consider dynamic obstacles, the obstacles and their positions are observed and subsequently, the velocity as well as the future position of the obstacle is predicted. Finally, a conservative bound around the moving obstacle is computed and the planner plans a collision-free path avoiding the moving object. Due to changes in the environment, replanning is done within a time budget. ITOMP can already deal with dynamic environments, yet there is no evaluation on runtime indicated. Moreover, fixed timings for the trajectory waypoints as well as the time budgets for planning are utilized, which can lead to trajectories which are not collision-free.

Another interesting approach to motion planning proposed by Brock and Khatib are elastic strips [2]. While ROS's motion planning pipeline as well as CHOMP and STOMP would have to employ frequent replanning to adapt to dynamic environments, the elastic strips idea plans a trajectory for the whole body of the robot and conforms it by means of reactive behavior. In order to obtain the trajectory a candidate path is chosen and the workspace volume of the robot representing its homotopic paths is computed. Subsequently, the best path is chosen with regards to being collision free and fulfilling given tasks. By means of potential fields the shape of the trajectory changes within the limits of the computed elastic tunnel simultaneously with its execution.

At the German Aerospace Center (DLR) whole body motion planning is implemented for their robot Rollin' Justin [5]. Like the elastic strips idea potential fields and reactive control are employed. Moreover, both approaches use torque control as well as the concept of tasks and task hierarchies. The motion planning applied for Rollin' Justin also considers the large number of degrees of freedom (DoF) and therefore, makes use of null space projection to get redundancy resolutions.

Since reactive behavior does not take the near future into account, failures occur with both elastic strips and the DLR motion planning and therefore, complete global replanning becomes necessary. Unlike both approaches our robots are not able to employ torque control. Instead, position control in terms of joint angles is used.

Integrating a multiresolutional approach into planning is not new. Not only is it used with regard to the environment representation, but also with respect to time. Both ideas base on the fact, that in a continuous and dynamic world uncertainty arises when moving. Hence, it is not worthwhile to make detailed plans for the far-future. Behnke utilizes both approaches for local multiresolution path planning for soccer robots [1]. In order to model the environment the method employs multiple robot-centered grids with different resolutions, and nests them hierarchically while connecting them through adjacencies. With regard to time the planning space gains another dimension in which the time moves only in one direction. Therefore, at the starting point the timesteps have a high resolution and gradually the precision decreases, that is the distance between timesteps gets larger. As it provides computational advantages and therefore, fastens computations, the multiresolution in time idea is included into the motion planning presented in this thesis.

In terms of motion planning for endeffectors He et al. also utilize the idea of a multiresolutional planning by proposing a multigrid CHOMP [7]. Unlike Behnke's approach, all of the timesteps are first computed in a low resolution and are subsequently upsampled by adding intermediate points between timesteps from the coarser resolution. While optimizing, the points from the lower resolution are fixed and only the new intermediate points are altered. They also utilize the fact, that the optimization algorithm converges faster, if an already near-optimal trajectory is given as initialization. As the latter is true for CHOMP and STOMP we also use previously planned trajectories as an initialization for further planning. Nevertheless, their idea of a multiresolutional trajectory does not appeal to us, because they have to plan multiple times, yet they end up with uniform spaced timesteps in a high resolution. Moreover, frequent replanning has to be executed in order to consider dynamic obstacles. In contrast, utilizing Behnke's approach provides a finished plan, which is only refined during replanning.

Overall, it is reasonable to adopt some parts of the proposed approaches for motion planning, which is for instance the multiresolution in time as well as the reusing of previously planned trajectories as an initialization.

3 Foundations

In order to solve motion planning problems, first the motion planning itself and the given problem has to be defined, followed by the favored output of the planning process. Since the goal is to not only find feasible and collision-free paths, but also include the kinematics of the robot, it is important to know kinematic chains as well as forward and inverse kinematics. With regard to implementing the motion planning, the body composition of the robots used for planning has to be known and because not every aspect of motion planning can be newly implemented, an already existing software framework has to be utilized.

In this chapter I will outline the basics of motion planning in robotics as well as point out some types of motion planning algorithms with different characteristics. In addition, I will present the utilized robots as well as describe the given software framework snd the currently employed motion planning for both robots.

3.1 Motion Planning in Robotics

In robotics one goal is to fulfill different kind of tasks, for which the robot or bodyparts of the robot have to execute motions. Hence, the problem is to find a collision-free motion for changing from one state of the robot to another state. In order to solve this problem, the robot model and its configurations, the environment, and the desired goal representation have to be defined.

While the goal of path planning is to find a feasible and collision-free path from the start configuration to the given goal configuration, motion planning considers the kinematics of the robot. This means that the trajectory resulting from the motion planning is not only feasible and collision-free, but also the velocities and accelerations are taken into account during planning. Moreover, it is possible to simultaneously optimize an objective function [4]. Overall, a resulting trajectory consists of a number of so-called waypoints in continuous space defining the configuration of the robot at a given timestep.

The configuration space represents all possible positions of the endeffector and its shape is determined by the degress of freedom of the robot model [4]. Moreover, the c-space consists of two sets C_{free} and C_{obs}. C_{free} represents all positions of the endeffector where no other bodypart is in collision, whereas C_{obs} defines the positions where collisions occur. An example of a configurations space and planning in the c-space is depicted in Fig. 3.1.

A kinematic chain is a mathematical model of the composition of the joints and rigid body parts of the robot with each one of them building up on the reference frame of its parent. Each child frame is defined by the translation as well as the rotation with regard to the reference frame. An example of a kinematic chain is shown in Fig. 3.2. The number of joints define the degrees of freedom for the kinematic chain, that is the number of independent parameters for the whole kinematic chain. The configuration of the robot is therefore defined by the values of this parameter vector.

In order to transform parts of the kinematic chain, that is moving the body parts, rigid-body transformations are utilized. A rigid-body transformation is a function, which maps

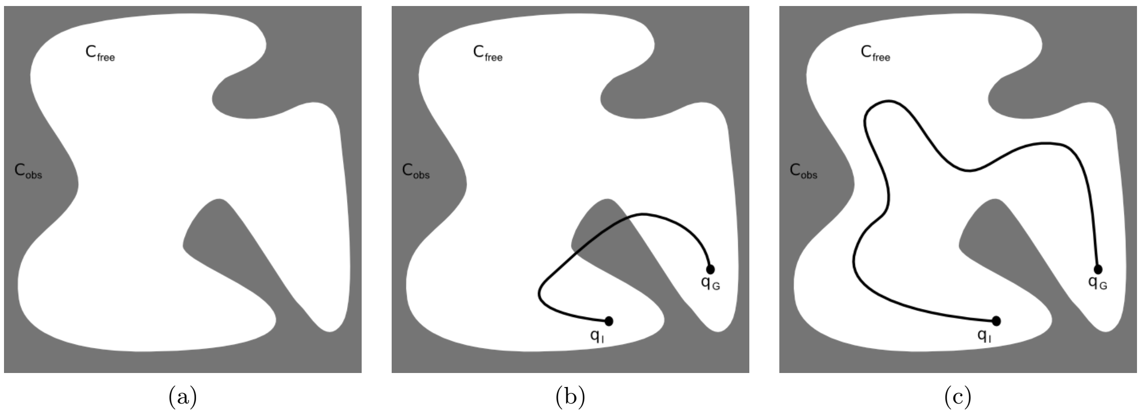


Figure 3.1: An example of a configuration space shown in (a) with an invalid path between a start configuration q_I and an end configuration q_G in (b) and a valid path shown in (c).

the robot parts from their body frame into the world frame, while preserving the distance between each pair of points of the robot part and also the orientation. Rigid-body transformations are affine transformations and thus, they are defined by a rotation matrix and a translation.

Overall, the relative movements of the joints and bodyparts are calculated by means of a sequence of rigid transformations from the start of the kinematic chain to the endeffector [9].

When planning motions, the goal is to find a path in joint space which translates into a desired path in the c-space. Since the endeffector positions are defined in c-space and the joint configurations are defined in joint space, mappings between both spaces are needed. When transforming the kinematic chain of the robot, two possibilities are given for determining either the position of the endeffector or the configuration of the joints. Those possibilities are forward kinematics and inverse kinematics, respectively.

When employing forward kinematics, kinematic equations of a robot are utilized to compute the position of the endeffector by means of joint angles for the given joints (cf. Fig. 3.3a). The kinematic equations used for computing the forward kinematics are based on rigid-body transformations. Calculating the forward kinematics might fail, when the parameters describe a configuration which is not feasible for the underlying kinematic chain of the robot. In order to detect whether or not a configuration is in C_{free}, the forward kinematics for the configuration is executed and then collision detection algorithms are utilized to detect collisions for the position of each body part.

In contrast to forward kinematics, inverse kinematics get the position of the endeffector as an input and subsequently calculate the parameters for the joints of the kinematic chain, which are needed to position the endeffector at the given pose (cf. Fig. 3.3b). Inverse kinematics can be utilized for computing goal configurations for motion planning, since defining goals for a task is more convenient in the c-space defining the position of the endeffector [4] [9].

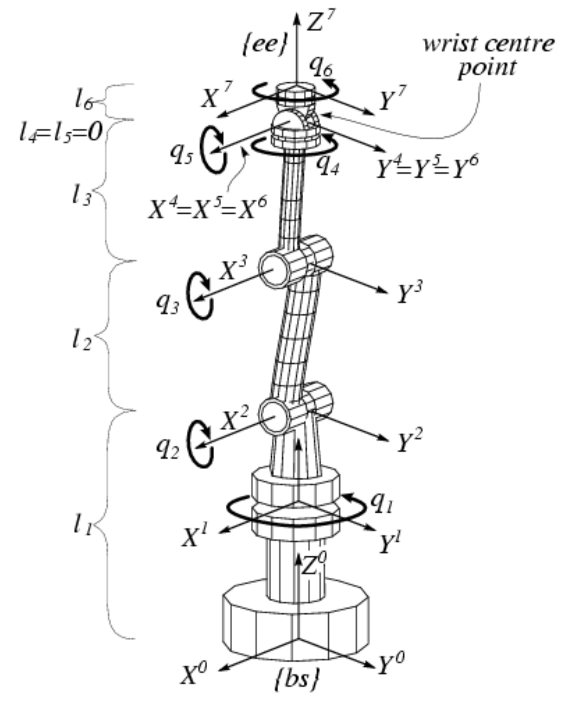
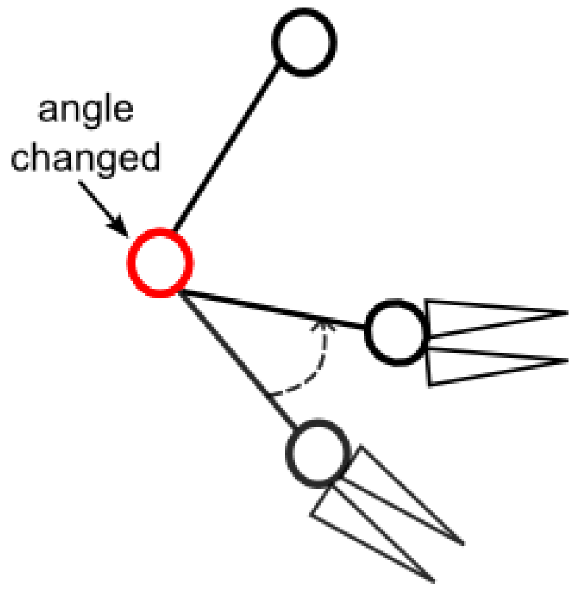


Figure 3.2: An example scheme of a kinematic chain with 6 degrees of freedom [15].

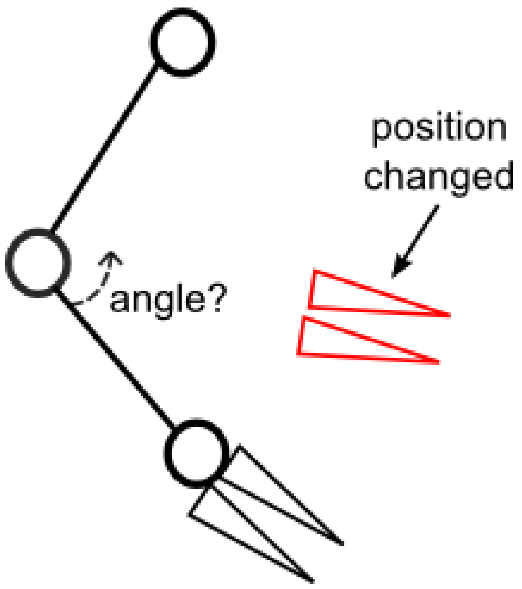
3.1.1 Types of Motion Planning Algorithms

In order to solve motion planning tasks, various types of motion planners can be utilized. It is possible to distinguish between two types of motion planning concepts being the decoupled approach and the direct approach. When using a decoupled approach, first a valid path is found in the configuration space and afterwards the path is refined in terms of time scaling, whereas the direct approach already includes this within its planning process [4].

For planning problems in a low-dimensional c-space, grid-based planners can be used to find feasible and collision-free paths. They employ a discretized version of the configuration space and utilize well-known search algorithms such as A* to find a valid path from the start configuration to the goal configuration. With an increasing number of dimensions, the number of gridcells grows exponentially and therefore, grid-based planners do not efficiently find solutions to the motion planning problems in high-dimensional c-spaces. Another method to perform motion planning are potential fields. The configuration space is translated into a gradient vector field with attractions towards the goal and repulsive forces directing the robot away from obstacles. Sampling-based planners use different methods for sampling the given c-space and then connect the samples with each other to find a path from the start configuration to the goal configuration. Since the planners only want to sample configurations in C_{free}, they utilize collision detection in order to determine whether a sampled configuration is in collision. When connecting two samples with each other, the connection is also tested for collisions. The sampling method is chosen due to the goal of planner and with regard to the desired exploration of the c-space. Established planners, such as the Rapidly-exploring Random Trees (RRT), use unidirectional or bidirectional trees for exploration [9]. In contrast to the other methods, optimization-based motion planning algorithms start with an initial configuration and iteratively enhance the initial guess until



(a) Forward Kinematics



(b) Inverse Kinematics

Figure 3.3: Forward kinematics are shown in (a). The angle of the red joint changes and accordingly the new endeffector position is computed.
Inverse kinematics are shown in (b). A new target position for the endeffector is given (red) and the joint angles needed to reach this position are calculated.

a feasible and collision-free path is found. Again collision detection is used to determine, whether a path is in C_{free}. Overall, sampling-based and optimization-based planners as well as potential fields are used for motion planning problems in high-dimensional c-spaces.

3.2 Domestic Service Robots Cosero and Dynamaid

Investigating domestic service robots, two fully autonomous robots were build from scratch at the Autonomous Intelligent Systems Group of the University of Bonn. In 2009 the domestic service robot called "Dynamaid" was built, succeeded by an improved version called "Cosero" two years later [17] [16].
Both robots follow an anthropomorphic design approach as they have to adapt to household environments, which are designed for human users. Due to Cosero being an enhanced version of Dynamaid, the two robots have a comparable assembly (cf. Fig. 3.4). They consist of a mobile basis with an omni-directional drive, a movable torso attached to a elevator, two anthropomorphic arms each equipped with a gripper, and a communication head. Moreover, they are both lightweight, as Dynamaid and Cosero only weigh $25kg$ and $30kg$, respectively. The robots are easily transportable and less dangerous than heavyweight robots, since they need less propulsive force for their motions. Due to their elevators they are able to move their torsos from a floor level up to a height of $180cm$. Therefore, it is possible not only to manipulate objects in high places, but also to grasp objects from the ground. In addition, their trunks are movable around their elevators, which leads to a larger workspace and concurrently improves the human-robot interaction with a user.

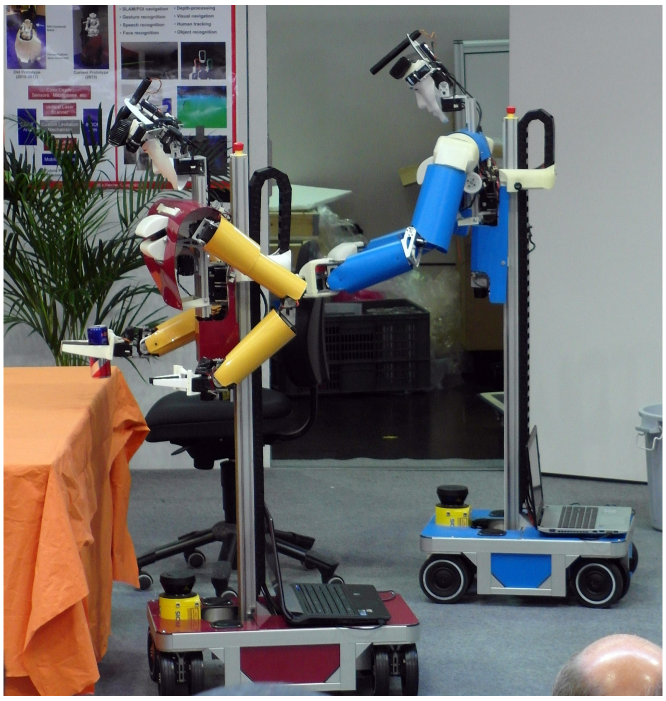
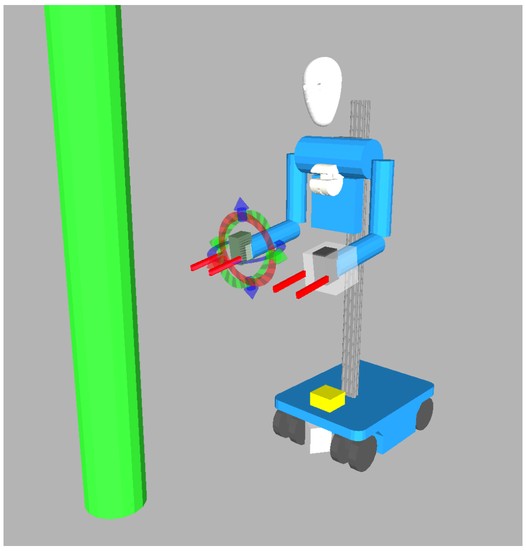


Figure 3.4: Domestic service robots Dynamaid (left) and Cosero (right) during the RoboCup German Open 2013.

Dynamaid and Cosero are equipped with a SICK S300 laser range finder, which is attached to the mobile basis. The laser range finder has an angle of beam of 270° and can measure distances up to 30m. Moreover, both robots have one Hokuyo URG-04LX attached beneath the trunk and one Hokuyo UTM-30LX in the chest. The Hokuyos are able to measure up to 10m with an angle of beam of 270°. Both laser range finders are movable. In case of the trunk laser the joint enables to measure not only horizontally, but also vertically. The other Hokuyo is used as a 3D-laser range finder. In addition, Cosero has a third Hokuyo floor laser in order to detect obstacles on the ground. Both robots feature a Kinect for Xbox 360 RGB-D camera on their head for perception, especially of the nearby environment.
Overall the robots possess a total of 32 joints, which are revolute except for the elevator joint, which is prismatic. Each of their anthropomorphic arms has 7 DOF and an attached endeffector with 2 DOF utilizing two parallel Festo FinGripper fingers on each opposing side. Due to the finray effect the fingers automatically bend inwards when grasping an object and therefore, including the object with the grippers The endeffectors can handle a load of 1kg and 1.5kg accordingly, as Cosero has stronger arms in terms of its propulsive force. Furthermore, being the improved version of Dynamaid, Cosero is more precise in its movements. The robots are actuated by Robotis Dynamixels of the types RX-64 and EX-106+ with maximum movement speeds of 64rpm and 91rpm, respectively, without load [14]. In contrast to using torque control, the actuators of the robots are adressed by position control. Although each joint holds a specified hardware limit, self collisions between the robots' parts are still possible.
Eventually each robot is controlled by the software running on a quad-core laptop being mounted on its basis.


(a) Visualization of Planning Environment

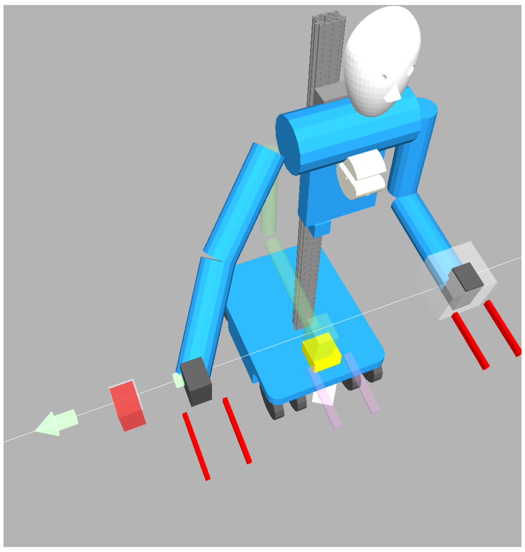

(b) Joint violation shown by planning components visualizer

Figure 3.5: The visualization of the robot model and a 3D mesh of an obstacle in Rviz including a marker to intervene with the right endeffector is shown in (a).
In (b) a joint violation occurring due to dragging the right endeffector to a position, which is not reachable, shown by the planning components visualizer is depicted.

3.3 Framework

The software of both robots is based on the open source middleware ROS created by Willow Garage [12]. With ROS it is possible to run various modules at a time. Those processes are called nodes and they are connected through a "master" or "name service". For communication between nodes messages written in an interface definition language can be send out. Those messages are published on so-called topics, to which other nodes can subscribe in order to read the published messages. Furthermore, one can define services, which get request messages from clients, run the requests through a predefined callback function and subsequently send back their computed response. In contrast to providing a whole runtime environment, ROS uses a number of smaller tools for several components to gain stability. In order to use the code in other projects as well, the software is bundled in libraries. ROS makes use of the code from other open source projects. Overall, numerous modules written by the ROS community for plenty different tasks are available. Since ROS is continuously developed further, new distributions of the middleware are provided at regular intervals. As a result, new modules are added, modules become obsolete and the structures and interfaces of existing modules are changed at times.
In addition to the modules included in ROS, various nodes have been implemented specifically for Cosero and Dynamaid. These include nodes for perception, navigation, simulation, graspplanning, manipulation and motion planning.

3.3.1 Visualization

ROS provides several tools for visualization. Out of these, the most important tool is Rviz, which can be used for 3D visualization. With Rviz it is possible to picture numerous display types such as axes, laser scans, point clouds, poses, robot models and markers displaying arbitrary primitive shapes. Hence, Rviz can be used for displaying states of a robot with its rigid body parts and the coordinate systems defined by its joints as well as the environment and how the robot perceives it. Moreover, the results and intermediate steps of executed modules, for example gridmaps or computed trajectories, can be shown. Therefore, Rviz is a very important tool for debugging, given that errors can be spotted more easily.
Combined with the planning components visualizer module Rviz is even more powerful, as the planning components visualizer permits intervening with the robot model as well as creating and moving 3D meshes of obstacles like poles (cf. Fig. 3.5a). Thus, the configuration of the robot joints can be changed by dragging the end effector while the module constantly checks if joint limits are met or singularities occur (cf. Fig. 3.5b). Accordingly, it is possible to define start and goal configurations of the robot and pass them on to modules dealing with motion planning. In addition, ground truth data of the previously created obstacles can be send and then processed, too. Altogether, both modules simplify the whole testing and debugging process regarding motion planning.

3.3.2 Planning environment

In order to utilize the body composition and properties of a given robot for computation, it is necessary to have an accurate robot model, which can be used by a lot of different modules. On this account, ROS makes use of the Unified Robot Description Format (URDF), which is an XML format and represents the robot model. An URDF file is composed of the links and joints of the robot, which are characterized by their origin, type, size and limits, respectively. Moreover, it can be used to define collision models for each link in terms of primitive shapes, which again form a complete collision model of the robot. Subsequently, this collision model is employed for checking self-collisions as well as collisions with obstacles.
In order to compute forward kinematics for instance, it is necessary to have the kinematic chain of the robot, as the URDF model is not designed for this task. The Kinematics and Dynamics Library (KDL) is able to parse a URDF robot model into a kinematic chain and moreover, the KDL solvers can be utilized directly for calculating forward kinematics or inverse dynamics.
Since both, Cosero and Dynamaid are custom-built, their kinematic properties can be obtained easily from CAD drawings. So far, robot models of both robots exist including their sensors, which can even gain artificial sensor measurements when being simulated.
The environment of the robot is perceived by the robot's laser range finders and the Kinect RGB-D camera. Accordingly, the measurements of the sensors are represented as a point cloud. The gained data then is transformed into a collision map, which is a 3D representation in terms of a list of occupied voxel cells including the size and orientation of the voxels. Eventually, the planning environment module engages in building a whole collision space, consisting of the current state of the robot as well as the environment, and moreover, monitoring the built collision space. After constructing a robot model out of the URDF file, the state of the robot is updated consecutively by means of present joint configurations. Those

are sent by a joint state publisher observing the actual robot state given by the planning components visualizer module, a simulation of the robot or the servos on the real robot. Furthermore, the planning environment module generates the collision map out of the given sensor data as well as manually inserted 3D meshes of objects and also updates the map by listening to incoming sensor measurements and changes to the ground truth data of the inserted meshes.
Both, the robot and the environment representations, are employed to check whether the robot configurations and planned motions are valid, as they might violate joint limits or constraints or yield to collisions or self-collisions. In order to speed up the self-collision test it is possible to disable checks between body parts when it is physically not feasible for them to collide. In addition, it can be checked if the environment is safe for movement, because it may become unsafe when important messages and information have not been received lately or sensor messages are outdated. Overall, the planning environment module provides the motion planning with numerous very important features needed for generating feasible plans.

3.3.3 Open Motion Planning Library

The Open Motion Planning Library (OMPL) [19] is a standalone library for sampling-based motion planning algorithms such as Rapidly-exploring Random Trees (RRT), Expansive Space Trees (EST) as well as Kinodynamic Planning by Interior-Exterior Cell Exploration (KPIECE) [18]. The library is integrated into ROS and only includes the implementations of the various motion planning algorithms. In order to connect OMPL with ROS an interface is given for configuring and utilizing the planners.
As an input to OMPL a robot model in URDF format is expected. Subsequently, a state space is automatically constructed out of the robot model, as OMPL is designed to work with any valid robot models and state spaces. By means of configuration files written in YAML one can define different planning groups consisting of a number of joints with regards to the URDF description of the robot. Also, self-collision checks between pairs of rigid body parts can be disabled, when it is physically impossible for those body parts to collide, in order to fasten the whole process of self-collision checking. Moreover, OMPL itself can be configured regarding the used planner types and the kinematics solver amongst other setups. Regarding the constructed state space, different state samplers can be used within the algorithms, for instance Gaussian, uniform or obstacle based samplers. Furthermore, state validity is completely defined by the programmer.
After defining the state space and picking one of the motion planning algorithms for computation, the planner needs a goal and is then called by other modules or by means of visualization tools like the planning components visualizer.

3.3.4 Current Motion Planning

To date, Dynamaid and Cosero are already able to use motion planning while grasping objects [10]. In order to find a goal for motion planning a grasp planner is employed, which provides a pre-grasp position for the endeffector of the robot and executes the grasp itself by means of a motion primitive. The motion planning module employs the given ROS planning pipeline including some changes with regards to the specific robot hardware. Therefore, it utilizes the KDL inverse kinematics solver to transform the pre-grasp position into a goal joint configuration. Subsequently, the goal configuration is passed on to the motion planning module, which then sends a planning request to the OMPL motion planning. Finally, the planned motion is executed by the robot and the desired object is grasped.

Kinodynamic Planning by Interior-Exterior Cell Exploration

One of the motion planning algorithms included in OMPL and moreover, the algorithm which is employed in the current motion planning module is KPIECE [18]. KPIECE is a sampling-based motion planner, which is designed to handle planning problems with complex dynamic constraints. It employs a discretization of the continuous state space in which it iteratively builds a tree of motions. The goal of the planner is to rapidly explore the state space. Hence, the discretization helps to decide which areas of the state space should be further explored. KPIECE supports a multilevel approach in order to efficiently detect less explored areas in the state space. The grids with low resolution rapidly show areas, which are not explored, yet, and the ones with high resolution are utilized for targeted exploration. The values of both, number of levels and resolution size, are constant. During the exploration, cells are only instantiated when needed and the preference for exploration is given to the boundaries of the cells.
The KPIECE algorithm is implemented in different variations such as Lazy Bi-directional KPIECE (LBKPIECE). In this version solution trajectories are searched not only from the start configuration on towards the goal configuration, but also in the other direction starting from the goal configuration. Therefore, LBKPIECE employs two trees for exploring the state space and moreover, lazy collision checking is employed to the solutions.
Although the multilevel approach is possible, the implementation of all KPIECE variations in OMPL only supports one level of discretization. Because KPIECE only finds feasible and collision-free plans, which are not optimized, it is necessary to apply post-processing to the solution trajectory. Accordingly, a trajectory filter is employed, which smooths the trajectory and shortens it by skipping dispensable waypoints.

4 STOMP: Stochastic Trajectory Optimization for Motion Planning

Concerning motion planning various algorithms for finding feasible and collision-free solutions exist. Stochastic Trajectory Optimization for Motion Planning (STOMP) by Kalakrishnan et al. is one out of a number of different motion planning algorithms [8]. It is based on another optimization-based motion planning algorithm called CHOMP by Ratliff et al. [13].

CHOMP drops the idea of first finding a feasible path and then optimizing it in order to remove redundant or jerky motions. It uses an initial trajectory, which does not have to be collision-free, as an initialization and then optimizes it by means of covariant gradient descent. Therefore, no post-processing has to take place to gain a collision-free and smooth trajectory. The goal of the optimization is to minimize the costs of the initial trajectory by means of iteratively updating it. The cost function consists of obstacle costs as well as velocity and acceleration costs for considering the smoothness of the trajectory. As CHOMP employs gradient descent, the cost function needs to be differentiable.

In contrast to CHOMP, STOMP follows the approach of stochastic trajectory optimization while adopting some of the benefits of CHOMP. Subsequently, I will present the STOMP algorithm in its details and then describe the implementation of STOMP. Finally, I will discuss some pros and cons of the STOMP algorithm.

4.1 Algorithm

Similarly to CHOMP, STOMP defines the given motion planning problem as an optimization problem. Accordingly, the goal is to find a trajectory, which minimizes the costs calculated by a predefined cost function.

As an input STOMP gets a start and a goal configuration $\boldsymbol{x}_{start}, \boldsymbol{x}_{goal} \in \mathbb{R}^J$, with J being the number of joints and therefore determining the dimension of the configuration space. The output of the algorithm is one trajectory vector $\boldsymbol{\theta} \in \mathbb{R}^N$ for each joint, with a fixed duration T and sampled into N waypoints. The waypoints are uniformly spaced with regard to a fixed discretization d. Both, the start and goal configurations as well as the whole trajectories, are specified in joint space and moreover, the start and the goal configurations stay fixed during the whole planning process. Besides predefining a cost function, an initial trajectory $\boldsymbol{\theta}$ for each joint has to be defined as a starting point for optimization.

For convenience, the STOMP algorithm will be presented only for one joint. In order to scale the algorithm to multiple joints, each step has to be executed for each joint separately. Thus, the computational complexity of STOMP scales linearly with the number of joints and dimensions, respectively.

Overall, the optimization problem STOMP wants to solve iteratively is defined by

$$\min_{\tilde{\boldsymbol{\theta}}} \mathbb{E}\left[\sum_{i=1}^{N} q(\tilde{\boldsymbol{\theta}}_i) + \frac{1}{2}\tilde{\boldsymbol{\theta}}^\top \mathbf{R}\tilde{\boldsymbol{\theta}}\right] \tag{4.1}$$

with $\tilde{\boldsymbol{\theta}} = \mathcal{N}(\boldsymbol{\theta}, \boldsymbol{\Sigma})$ being a noisy joint parameter vector with mean $\boldsymbol{\theta}$ and covariance $\boldsymbol{\Sigma}$. $q(\tilde{\boldsymbol{\theta}}_i)$ is a predefined cost function calculating the costs for each state in $\tilde{\boldsymbol{\theta}}$. Also, $\boldsymbol{\theta}^\top \mathbf{R}\boldsymbol{\theta}$ describes the sum of squared accelerations along the trajectory with $\mathbf{R}$ being a positive semi-definite matrix representing the control costs. In order to derive $\mathbf{R}$ a matrix $\mathbf{A}$ is defined, which represents a discrete filter for calculating the accelerations $\ddot{\boldsymbol{\theta}}$ when multiplied by the position vector $\boldsymbol{\theta}$. Eventually, $\mathbf{A}$ is defined as

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & & 0 & 0 & 0 \\ -2 & 1 & 0 & \dots & 0 & 0 & 0 \\ 1 & -2 & 1 & & 0 & 0 & 0 \\ & \vdots & & \ddots & & \vdots & \\ 0 & 0 & 0 & & 1 & -2 & 1 \\ 0 & 0 & 0 & \dots & 0 & 1 & -2 \\ 0 & 0 & 0 & & 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

by means of finite differencing [6]. Hence, it holds

$$\ddot{\boldsymbol{\theta}} = \mathbf{A}\boldsymbol{\theta} \tag{4.3}$$
$$\ddot{\boldsymbol{\theta}}^\top \ddot{\boldsymbol{\theta}} = \boldsymbol{\theta}^\top (\mathbf{A}^\top \mathbf{A})\boldsymbol{\theta}. \tag{4.4}$$

Consequently, $\mathbf{R}$ is selected as $\mathbf{R} = \mathbf{A}^\top \mathbf{A}$ in order to represent the control costs.

In constrast to CHOMP, which effectively optimizes the non-stochastic version of Eqn. 4.1 by means of covariant gradient descent, STOMP now attempts to solve the defined optimization problem by means of a stochastic optimization method. As a result it is possible to minimize the predefined cost function $q(\tilde{\boldsymbol{\theta}})$, even though it might not be differentiable and therefore, no gradients are available for this function.

Instead of calculating the derivatives of the cost function, STOMP only estimates the gradient. Accordingly, when taking the gradient of the expectation in Eqn. 4.1 with respect to $\tilde{\boldsymbol{\theta}}$, it holds

$$\nabla_{\tilde{\boldsymbol{\theta}}} \left(\mathbb{E}\left[\sum_{i=1}^{N} q(\tilde{\boldsymbol{\theta}}_i) + \frac{1}{2}\tilde{\boldsymbol{\theta}}^\top \mathbf{R}\tilde{\boldsymbol{\theta}}\right]\right) = 0. \tag{4.5}$$

After differentiating the expectation of the sum of squared accelerations along the trajectory, we get

$$\mathbb{E}(\tilde{\boldsymbol{\theta}}) = -\mathbf{R}^{-1}\mathbb{E}\left(\nabla_{\tilde{\boldsymbol{\theta}}}\left[\sum_{i=1}^{N} q(\tilde{\boldsymbol{\theta}}_i)\right]\right), \tag{4.6}$$

which can also be rewritten as $\mathbb{E}(\tilde{\boldsymbol{\theta}}) = -\mathbf{R}^{-1}\delta\hat{\boldsymbol{\theta}}_G$, with $\delta\hat{\boldsymbol{\theta}}_G$ defining the gradient estimate

$$\delta\hat{\boldsymbol{\theta}}_G = \mathbb{E}\left(\nabla_{\tilde{\boldsymbol{\theta}}}\left[\sum_{i=1}^{N} q(\tilde{\boldsymbol{\theta}}_i)\right]\right). \tag{4.7}$$

Subsequently, by means of probability matching as well as path integral reinforcement learning, $\delta\hat{\boldsymbol{\theta}}_G$ is paraphrased as

$$\delta\hat{\boldsymbol{\theta}}_G = \int \delta\boldsymbol{\theta}\ d\mathbf{P} \tag{4.8}$$

being the expectation of $\delta\boldsymbol{\theta}$ under the probability metric $\mathbf{P}$, with $\delta\boldsymbol{\theta}$ being the noise in $\tilde{\boldsymbol{\theta}}$. Furthermore, it holds

$$\mathbf{P} = exp\left(-\frac{1}{\lambda}S(\tilde{\boldsymbol{\theta}})\right) \tag{4.9}$$

$$S(\tilde{\boldsymbol{\theta}}) = \left[\sum_{i=1}^{N} q(\tilde{\boldsymbol{\theta}}_i)\right] \tag{4.10}$$

with $S(\tilde{\boldsymbol{\theta}})$ being the state dependent cost defined on the trajectory. As a result, the gradient estimate is defined as

$$\delta\hat{\boldsymbol{\theta}}_G = \int exp\left(-\frac{1}{\lambda}S(\tilde{\boldsymbol{\theta}})\right)\delta\boldsymbol{\theta}\ d(\delta\boldsymbol{\theta}). \tag{4.11}$$

Since STOMP is based on path integral stochastic optimal control [20], its further calculations are quite similar. In the path integral stochastic optimal control framework the goal is to find optimal controls, which minimize a performance criterion. Therefore, these controls are calculated for each state $\boldsymbol{x}_{ti}$ as $\delta\hat{\boldsymbol{u}} = \int p(\boldsymbol{x})\delta\boldsymbol{u}$. In this equation $\delta\boldsymbol{u}$ are the sampled controls and $p(\boldsymbol{x})$ describes the probability of each trajectory $\boldsymbol{\tau}_i = (\boldsymbol{x}_{ti}, \ldots, \boldsymbol{x}_{tN})$. Similarly to STOMP, the probability function is defined as $p(\boldsymbol{x}) = exp(-S(\boldsymbol{\tau}_i))$ with $S(\boldsymbol{\tau}_i)$ calculating the costs for the whole trajectory $\boldsymbol{\tau}_i$. Hence, $p(\boldsymbol{x})$ is inversely proportional to the cost $S(\boldsymbol{\tau}_i)$, which leads to trajectories with higher cost contributing less to the optimal controls than trajectories with lower cost. Being a multistage optimization, this is computed for each state $\boldsymbol{x}_{ti}$. Finally, after updating the controls $\boldsymbol{u} = \boldsymbol{u} + \delta\hat{\boldsymbol{u}}$, new paths are generated [20].

Regarding the STOMP algorithm, the cost $S(\boldsymbol{\theta}_i)$ is defined as a local cost by $S(\boldsymbol{\theta}_i) = q(\boldsymbol{\theta}_i)$. This means that the cost function only depends on the current state $\boldsymbol{\theta}_i$ in contrast to considering the whole trajectory for calculating the cost of each state. The resulting algorithm for STOMP is shown in Alg. 1.

Before generating the noisy trajectories out of an initial trajectory plus noise, the noise $\boldsymbol{\epsilon}$ is sampled from a zero mean normal distribution with covariance $\boldsymbol{\Sigma}_{\boldsymbol{\epsilon}} = \mathbf{R}^{-1}$ (cf. Fig. 4.1). Due to the noise samples $\boldsymbol{\epsilon}$ accordingly having low control costs $\boldsymbol{\epsilon}^\top\mathbf{R}\boldsymbol{\epsilon}$, the control costs of the noisy trajectories are not relevantly affected by the added noise, while the exploration of the configuration space can still take place. For the same reason, the addition of the noise does not lead to the trajectory diverging from the start and goal configurations and moreover, it might be possible to execute the noisy trajectories on real systems without further problems.

```
Given:
- Start and goal positions $x_0$ and $x_N$
- An initial 1-D discretized trajectory vector $\boldsymbol{\theta}$
- A state-dependent cost function $q(\boldsymbol{\theta}_i)$
Precompute:
- $\mathbf{A}$ = finite difference matrix (see Eqn. 4.2)
- $\mathbf{R}^{-1} = (\mathbf{A}^\top\mathbf{A})^{-1}$
- $\mathbf{M} = \mathbf{R}^{-1}$, with each column scaled such that the maximum element is $\frac{1}{N}$
repeat
    for $k = 1$ to $K$ do
        // Create K noisy trajectories
        $\tilde{\boldsymbol{\theta}}_k = \boldsymbol{\theta} + \boldsymbol{\epsilon}_k$ with $\boldsymbol{\epsilon}_k = \mathcal{N}(0, \mathbf{R}^{-1})$
    for $k = 1$ to $K$ do
        // Compute state costs and probabilities for each timestep $i$
        for $i = 0$ to $N$ do
            $S(\tilde{\boldsymbol{\theta}}_{k,i}) = q(\tilde{\boldsymbol{\theta}}_{k,i})$
            $P(\tilde{\boldsymbol{\theta}}_{k,i}) = \frac{e^{-\frac{1}{\lambda}S(\tilde{\boldsymbol{\theta}}_{k,i})}}{\sum_{l=1}^{K}[e^{-\frac{1}{\lambda}S(\tilde{\boldsymbol{\theta}}_{l,i})}]}$
        // Calculate noisy parameter update
        for $i = 0$ to $N$ do
            $[\delta\tilde{\boldsymbol{\theta}}]_i = \sum_{k=1}^{K} P(\tilde{\boldsymbol{\theta}}_{k,i})[\boldsymbol{\epsilon}_k]_i$
    // Smooth noisy parameter update and update trajectory
    $\delta\boldsymbol{\theta} = \mathbf{M}\delta\tilde{\boldsymbol{\theta}}$
    $\boldsymbol{\theta} = \boldsymbol{\theta} + \delta\boldsymbol{\theta}$
    // Calculate trajectory cost
    $Q(\boldsymbol{\theta}) = \sum_{i=1}^{N} q(\boldsymbol{\theta}_i) + \frac{1}{2}\boldsymbol{\theta}^\top\mathbf{R}\boldsymbol{\theta}$
until Convergence of $Q(\theta)$;
```

Algorithm 1: The STOMP algorithm in 1-D

After creating K noisy trajectories, first the costs for each timestep $S(\tilde{\boldsymbol{\theta}}_{k,i})$ as well as the probabilities for each trajectory $P(\tilde{\boldsymbol{\theta}}_{k,i})$ are computed. When computing the probabilities, the parameter λ adjusts the impact of the exponentiated costs and moreover, it is automatically optimized for each waypoint by employing the minimum and maximum costs for the present waypoint over all K noisy trajectories. It holds

$$e^{-\frac{1}{\lambda}S(\tilde{\boldsymbol{\theta}}_{k,i})} = e^{-h\frac{S(\tilde{\boldsymbol{\theta}}_{k,i}) - min\ S(\tilde{\boldsymbol{\theta}}_{k,i})}{max\ S(\tilde{\boldsymbol{\theta}}_{k,i}) - min\ S(\tilde{\boldsymbol{\theta}}_{k,i})}} \tag{4.12}$$

with h being a constant. In the next step of the algorithm, a noisy update is computed for each time step by means of building the probability-weighted convex combination of each noisy trajectory at this waypoint. Subsequently, the noisy update is smoothed by multiplying it by the scaled inverse control cost matrix $\mathbf{M}$, which has $\frac{1}{N}$ as the maximum element in each column. As a result, none of the updated waypoints crosses the range determined by the exploration by means of the noisy trajectories and furthermore, the smoothness of the updated trajectory is assured. Eventually, the trajectory is updated by means of the smoothed update. Overall, the algorithm iteratively optimizes the initial trajectory until the trajectory cost $Q(\boldsymbol{\theta})$ converges.

The cost function utilized in the STOMP algorithm consists of different parts such as obstacle costs q_o, constraint costs q_c, and torque costs q_t. In summary, the cost function is defined as

$$q(\boldsymbol{\theta}) = \sum_{t=1}^{T} q_o(\boldsymbol{\theta}_t) + q_c(\boldsymbol{\theta}_t) + q_t(\boldsymbol{\theta}_t) \tag{4.13}$$

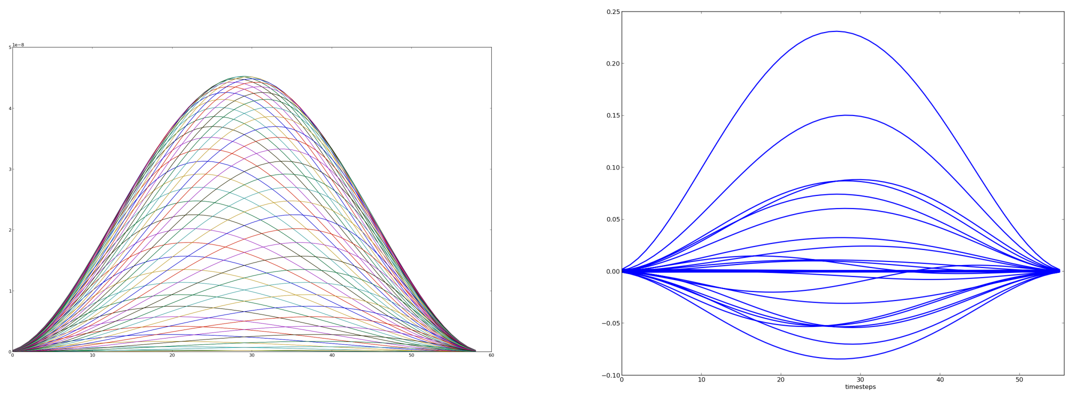
for the number of timesteps T. In order to calculate the obstacle costs a signed Euclidean Distance Transform (EDT) [21] is computed by means of a binary voxelgrid. Consequently, the function $d(x)$ based on the signed EDT returns the distance of x to the boundary of the closest obstacle. If x lies within the obstacle, $d(x)$ returns a negative value and otherwise, the value is positive. This also means, that discretized information is given about the penetration depth as well as distance and contact. The robot body $\mathcal{B}$ is approximated by a set of overlapping spheres $b \in \mathcal{B}$, whose surface should be at least ϵ distant to the closest obstacle. Hence, the origin of the sphere should be $\epsilon + r$ away from the closest obstacle with r being the radius of the sphere. Accordingly, it holds

$$q_o(\boldsymbol{\theta}_t) = \sum_{b \in \mathcal{B}} max(\epsilon + r_b - d(x_b), 0)\|\dot{x_b}\| \tag{4.14}$$

with r_b being the radius and x_b being the origin of the sphere b. Multiplying $\|\dot{x_b}\|$, which is the magnitude of the workspace velocity of the sphere b, avoids trajectories in which the robot moves rapidly through high-cost areas in order to lower the costs. The constraint costs q_c are defined by

$$q_c(\boldsymbol{\theta}_t) = \sum_{c \in C} |v_c(\boldsymbol{\theta}_t)| \tag{4.15}$$

with C being a set of a constraints and v_c calculating the magnitude of constraint violation for each constraint $c \in C$. Overall, constraints such as the endeffector position or orientation are considered by increasing the costs for each constraint violation. Similarly, the



(a) Inverse control cost matrix

(b) Sampled noise

Figure 4.1: (a) Each curve represents a column and a row of the symmetric inverse control cost matrix $\mathbf{R}^{-1}$. (b) 20 random noise samples $\boldsymbol{\epsilon} = \mathcal{N}(0, \mathbf{R}^{-1})$.

minimization of torques is taken into account by adding the torque cost

$$q_t(\boldsymbol{\theta}_t) = \sum_{t=0}^{T} |\boldsymbol{\tau}_t| dt \tag{4.16}$$

with the torque being $\boldsymbol{\tau}_t = f(\boldsymbol{x}_t, \dot{\boldsymbol{x}}_t, \ddot{\boldsymbol{x}}_t)$ with $\boldsymbol{x}_t = \boldsymbol{\theta}_t$. The velocities and accelerations are again obtained by finite differentiation.

Apart from these costs a cost for penalizing the violation of joint limits can be added to the cost function. Nevertheless, already dealing with the joint limits during the exploration of the configuration space by means of noisy trajectories is preferred. Due to the updated trajectory being a convex combination of the noisy trajectories, it stays within the given joint limits. Moreover, because of the trajectories being smooth the updated trajectory does not approach the joint limit with high speed and therefore, collisions with the joint limits are not likely.

4.2 Implementation

CHOMP is integrated into the Arm Navigation Stack of the ROS middleware and updated at times. Unfortunately, only an old implementation of STOMP from 2011 is available online and had to be updated and integrated manually. Due to ROS frequently releasing new distributions since March 2010, the standalone STOMP code had to be changed in order to run with ROS Groovy Galapagos. Still, the alterations only affect the interfaces and ROS classes used by STOMP and therefore, the key classes stay the same. In Fig. 4.2 the overall structure of the classes in the implementation of STOMP is depicted. Furthermore, important function calls are marked by the blue arrows. Collectively, the STOMP motion planner module is addressed by a service call, which is depicted in the figure in terms of the red arrows. Subsequently, I will outline the implementation of those main classes.

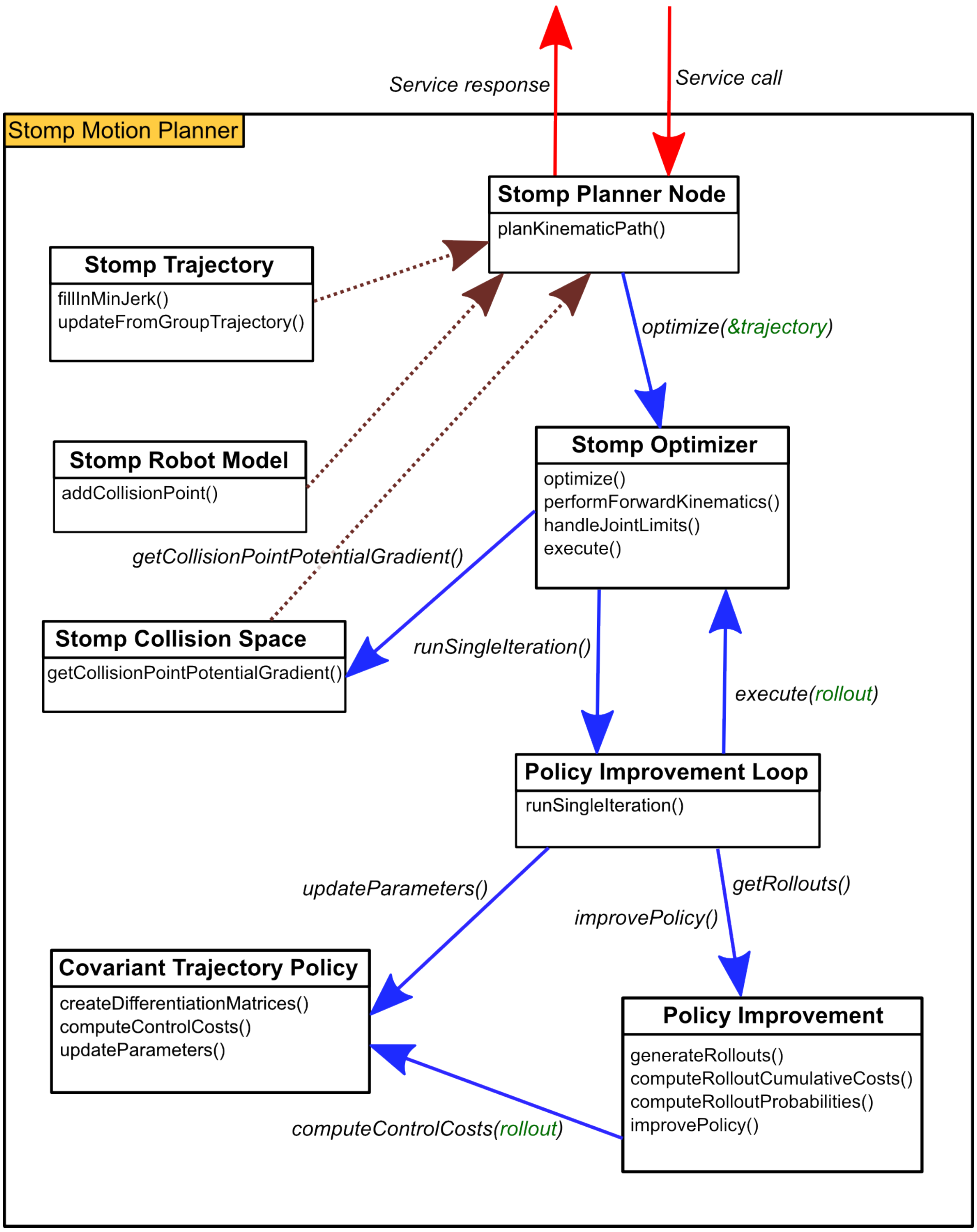


Figure 4.2: The structure of the main classes in the STOMP implementation including important function calls. The red arrows represent service calls, the blue arrows depict function calls and the dotted arrows represent, that entities of those classes are created in the pointed class.

```
# This service contains the definition for a request to the motion
# planner and the output it provides

MotionPlanRequest motion_plan_request

---

# A solution trajectory, if found
arm_navigation_msgs/RobotTrajectory trajectory

# The corresponding robot state
arm_navigation_msgs/RobotState robot_state

# Planning time
duration planning_time

# Error code - encodes the overall reason for failure
arm_navigation_msgs/ArmNavigationErrorCodes error_code

# More detailed error codes (optional) - encode information about each point
    in the returned trajectory
arm_navigation_msgs/ArmNavigationErrorCodes[] trajectory_error_codes
```

Sourcecode 4.1: GetMotionPlan Service

4.2.1 Stomp Planner Node

The `Stomp Planner Node` is the core of STOMP. It not only runs the motion planner, but also implements the interface between STOMP and the other software components of ROS. It processes the incoming planning requests sent by other modules and sends out a response message including the computed trajectory and additional information e.g. for visualization.

When starting the STOMP node, it first reads various parameters such as the trajectory duration or the trajectory discretization, which are defined within a config file amongst others. Moreover, the robot description defined by an URDF file is read and a `Stomp Robot Model` as well as a `Stomp Collision Space` is built (cf. 4.2.3). Finally, the service for receiving and processing motion planning requests is advertised and the `Stomp Planner Node` starts to wait for a service call.

By the time a motion planning request is received, the `plan_kinematic_path` callback function is invoked. While the callback function is processing the request, it cannot be called again with another request. After the callback function is returned, it is possible to solve a new motion planning problem.

The message which is sent to STOMP is a `GetMotionPlan` service request included in the `Arm Navigation Msgs` package of ROS. It consists of a `MotionPlanRequest` as well as a service response consisting of various other messages and particularly a `RobotTrajectory` message which includes the solution trajectory (cf. Source 4.1).

After receiving a service call, the goal constraints are translated into a goal configuration in joint space. If the goal is not well-defined and therefore not valid, the planner is aborted. Moreover, the goal configuration is fixed as needed in order to move to the shortest angular distance with regard to wrap-around joints. In contrast to the start configuration, the goal constraints only include values for the joints in the planning group, which is retrieved from the motion planning request and includes the names of the joints for which motion planning should be executed. Therefore, in order to set the goal configuration for the whole robot the start configuration is copied and the values of the planning group joints are replaced by the goal configuration.

In the next step, the collision points in the robot model are populated with the the joints listed in the planning group. Moreover, the start state of the robot is used as an input for creating a distance field from the collision space. Eventually, a trajectory of the type `Stomp Trajectory` (cf. 4.2.2) is created by means of the previously read in trajectory duration and discretization. In doing so, the start and end point of the trajectory are set and furthermore, the residual waypoints are initialized.

Subsequently, an entity of the `Stomp Optimizer` (cf. 4.2.4) is created getting numerous arguments as an input such as the previously initialized trajectory, the robot model, the planning group and the collision space amongst others. Accordingly, the actual optimization within the optimizer is executed. Since the trajectory is passed-by-reference, it is changed within the optimizer and therefore, contains the optimized trajectory after the optimization converged. Consequently, the trajectory is used to fill in the service response regarding the positions in joint space for each joint and the time from start for every waypoint. Along the way violations of the joint velocity limits are checked and hence, the durations between two timesteps are adapted if required. Finally, the service response is sent back to the service client, which can further process the computed trajectory.

4.2.2 Stomp Trajectory

As the goal of motion planning is to find a feasible, collision-free, and smooth trajectory, it is necessary to define a trajectory representation when implementing motion planning algorithms. With regard to STOMP the implementation needs particular attributes.

STOMP distinguishes between so-called full trajectories and group trajectories. Full trajectories are constructed given a trajectory duration and a trajectory discretization. Both parameters are predefined in a configuration file and they determine the number of timesteps for the trajectory. The actual waypoints are saved in a matrix of dimension $J \times N$ with J being the number of joints and N being the number of timesteps. The unit of the waypoint values is rad as the trajectories are specified in joint space.

In order to build an initial trajectory, at the beginning start and goal configurations are set. Subsequently, the intermediate waypoints are filled by means of B$\acute{e}$zier splines. It is assumed that the velocities and accelerations are zero at the start and end point. Accordingly, the spline coefficients are calculated for each joint and then utilized to compute the joint positions at each timestep. As a result the initial trajectories for each joint are short and smooth (cf. Fig. 4.3).

Group trajectories are based on the previously described full trajectories. They are constructed by getting the full trajectory as an input as well as the planning group and a differentiation rule length. The latter refers to the predefined finite differencing filters for determining the velocity and acceleration of a given waypoint. In order to also apply those filters to the start and end points, the group trajectory possesses a number of additional points on either side of the trajectory. Because of the start and goal configuration being static, there is no movement and therefore no slope before and after the trajectory. Thus, the additional points are filled with the start configuration and the goal configuration,

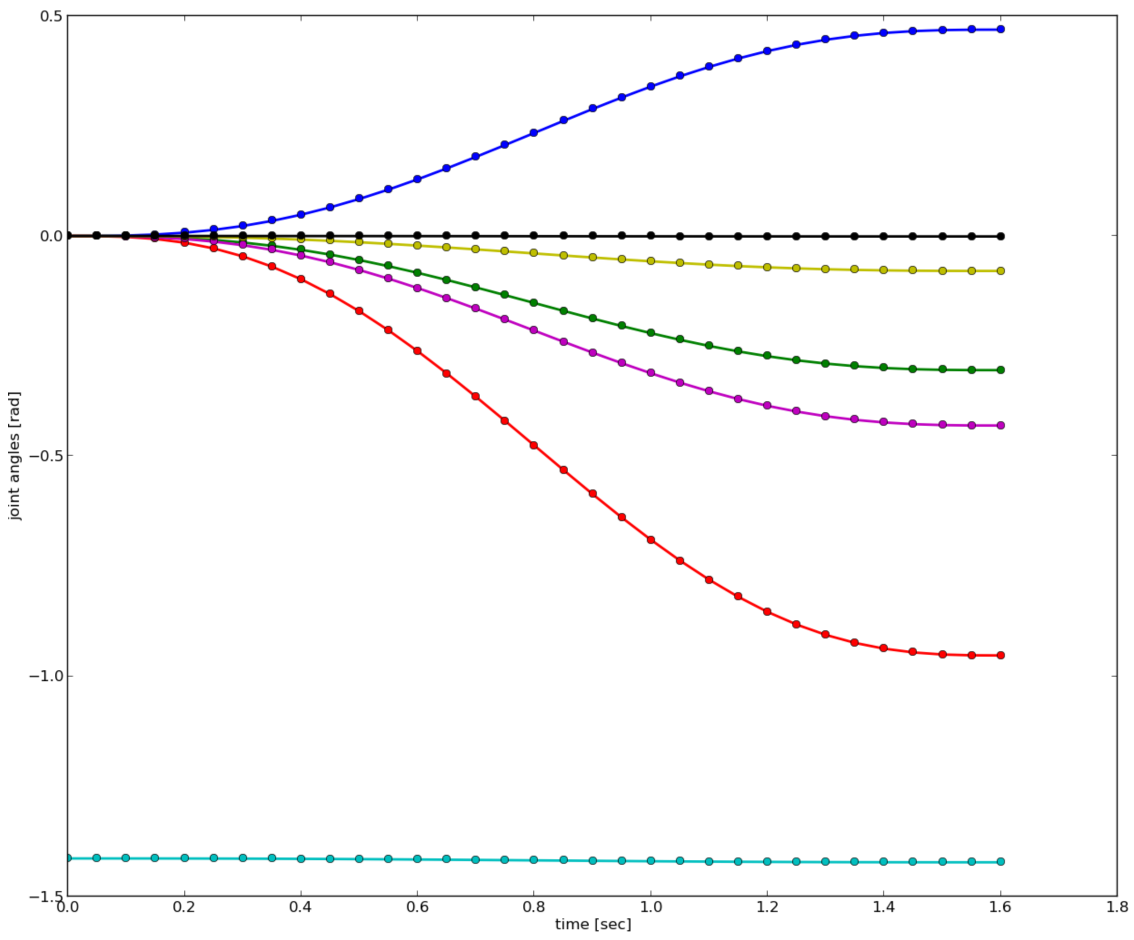


Figure 4.3: A set of uniformly spaced initial full trajectories for the joints of interest, that is the 7 joints of the robot arm.

respectively. While the full trajectory includes all joints of the robot, the group trajectory only includes the joints of interest, which are specified by the planning group. As a result, the matrix storing the group trajectory is of the dimensions $D \times P$ with D being the dimension of the c-space and the number of the joints in the planning group. Also, P is the size of each group trajectory with $P = N + 2 * (diff\_rule\_length - 1)$ and $diff\_rule\_length$ being the size of the finite differentiation filter. In order to map the full trajectory indices to the group trajectory indices a structure called full trajectory index is maintained. In contrast to the standard trajectory, the group trajectory not only has a start and an end point, but also holds a start index and end index. Those additionally defined indices determine the interval in which the actual optimization takes place. Since the start and goal configuration have to be fixed, the number of free variables F is limited to $F \leq N - 2$. Furthermore, the start and end indices can be placed arbitrarily within the trajectory while following the constraint that the start index is set chronologically before the end index. An overview of both, full trajectory and group trajectory, for one dimension is depicted in Fig. 4.4.

While the full trajectory specifies movements for the whole robot and is therefore sent out as a response regarding the requested motion planning problem, the group trajectory

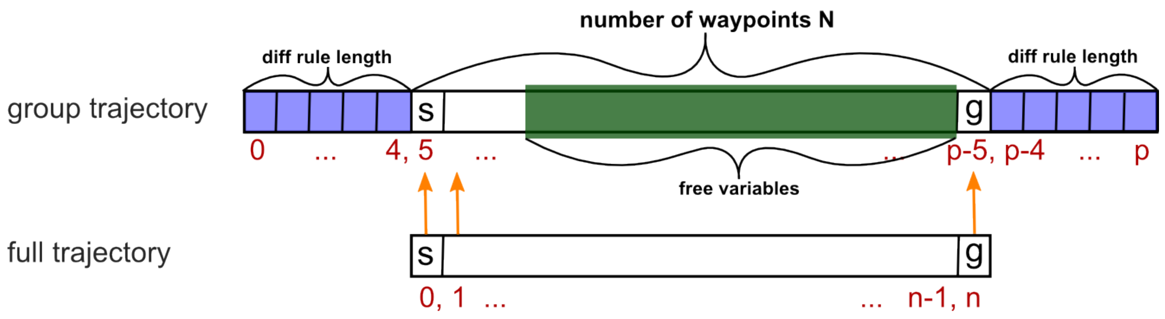


Figure 4.4: The structure of a 1-dimensional group trajectory and full trajectory in comparison. The group trajectory contains additional waypoints prior to and after the full trajectory. Also a start and an end index can be set to contain the free variables for optimization.

is only utilized for optimization purposes. This means that the group trajectory is only existing within the optimizer. Consequently, functions are defined to translate between the representations by utilizing the full trajectory index.

4.2.3 Collision Space and Robot Model

Although ROS provides a planning environment, the STOMP implementation needs its own representations of the environment and the robot with functions pursuant to it in order to execute the motion planning.

One of thoses classes is the `Stomp Collision Space` providing the representation of the environment in terms of a signed EDT. As the signed EDT a propagation distance field implemented in ROS is used, which employs a vector propagation method. It is created each time a service request is sent to the `Stomp Planner Node` and subsequently, the distance field is filled by means of the collision map created and maintained by the ROS planning environment. Moreover, bodies as well as bodyparts of the robot can be directly added to the distance field within the `Stomp Collision Space`. The `Stomp Collision Space` also provides functions which output the distance to the nearest obstacle for a given input point taken from the distance field. These functions are utilized in the `Stomp Optimizer` class in order to calculate the obstacle cost for each timestep of a trajectory.

Another class in STOMP which deals with the given robot model is the `Stomp Robot Model`. It includes structures for joints and for the planning group and contains the robot model in terms of a KDL tree. The KDL tree is build from the imported URDF file when constructing the `Stomp Robot Model` in the `Stomp Planner Node`. Due to the joints of the URDF and the KDL tree having a different order, functions are provided to convert the indices with regard to the other representation. Apart from this, the URDF file is used to generate a number of collision points for each link from the planning group as well as potentially attached objects to the robot. The `Stomp Robot Model` provides a KDL forward kinematics solver, which is together with the generated collision points of the robot model employed for detecting possible collisions.

4.2.4 Optimizer

In order to employ the optimization algorithm of STOMP eventually the `Stomp Planner Node` creates an instance of the `Stomp Optimizer` and passes various required objects such as the collision space, the robot model, and most important the previously built initial full trajectory. When initializing the optimizer, the group trajectory is built from the passed full trajectory by means of the joints defined in the planning group.

Subsequently, in the invoked optimize function a `Policy Improvement Loop` is created, which basically controls the optimization process. Still, the optimizer checks after each iteration whether the optimization has converged or no collision-free trajectory was found. The optimization converges when the trajectory costs do not decrease further through optimization and the last trajectory is collision-free. As a minimum number of iterations is predefined in the configuration of the planner, at least this number of iterations has to be performed in spite of already finding a collision-free trajectory with low costs. Similarly, a maximum number of iterations is defined, after which the planner aborts if no collision-free trajectory was found until then. Due to the full trajectory being passed-by-reference, it is automatically filled with the last and best group trajectory.

Apart from the optimize function, the `Stomp Optimizer` also deals with calculating the state costs for a given trajectory. This is done within an execute function, which adds up various costs such as the obstacle, torque, and constraint costs. Before calculating any costs, a function for handling the joint limits is called. Each waypoint is checked regarding the maximum and minimum joint limits for each joint and if a joint limit violation is detected, a cost value is added to the whole trajectory to avoid the violation. When there are no joint limit violations left, the overall state costs are computed for each waypoint. After calculating the torque cost and evaluating possible constraints, the obstacle costs are computed in terms of the equations 4.14, 4.15 and 4.16, respectively.

In order to calculate the obstacle costs, forward kinematics is performed by the KDL forward kinematics solver included in the `Stomp Robot Model`. Moreover, each collision point of the robot model is checked against the distance field from the `Stomp Collision Space` resulting in the collision point potential. Also, the magnitude of the workspace velocity of each collision point sphere is determined and multiplied by the collision point potential in order to get the obstacle costs. In the end, each of the cost segments is weighted and summed up to the final state costs (cf. Eqn. 4.13).

4.2.5 Policy Improvement Loop

Invoked by the `Stomp Optimizer` the `Policy Improvement Loop` manages the overall optimization process. For each iteration the `Policy Improvement Loop` follows the main steps of the STOMP algorithm by further utilizing the `Policy Improvement`, the `Covariant Trajectory Policy`, and also the `Stomp Optimizer` itself.

After getting newly generated rollouts, it fetches the state costs for each rollout by calling the execute function in the optimizer and subsequently, utilizes the policy improvement entity to also get the control costs. It then invokes functions for calculating the parameter updates and using them on the initial or last optimized trajectory.

Finally, it gets the updated, noise-less rollout and adds it to the rollout pool. This is repeated until the optimizer stops the planning due to converging or due to exceeding the maximum number of iterations.

4.2.6 Policy Improvement

The `Policy Improvement` includes a structure for trajectory rollouts consisting of the trajectory parameters, sampled noise, state and control costs as well as total costs and probabilities.

Policy improvement provides a function for generating new rollouts by sampling noise and adding it to the last optimized trajectory. Furthermore, rollouts from previous iterations are sorted by their total costs and a predefined number of rollouts with low costs is reused.

In order to improve the initial trajectory, parameter updates have to be determined. Hence, the total costs of the waypoints of each rollout are calculated by adding the state costs and the control costs. While the state costs are passed by the `Policy Improvement Loop`, the control costs have to be requested from the `Covariant Trajectory Policy`. In the next step, the probability of every waypoint is computed in order to generate the probability-weighted convex combination of all rollouts. Subsequently, the parameter updates are calculated by first determining the noisy updates and then reweighting and smoothing them.

4.2.7 Covariant Trajectory Policy

The `Covariant Trajectory Policy` mainly calculates the control costs and initializes the finite differencing matrix.

In order to create the finite differencing matrix (cf. Eqn. 4.2), the `Covariant Trajectory Policy` imports discrete filters for each order of derivative with predefined values, which are taken from the work of Fornberg on finite differencing [6]. As the trajectory resolution can be chosen arbitrarily, the filters are divided by the resolution to the power of the order of derivative. Eventually, the control cost matrix as well as the inverse control cost matrix is build from the finite differencing matrix.

Apart from providing a function for calculating the control costs for a given trajectory, the `Covariant Trajectory Policy` also updates the initial trajectory in each iteration given the joint parameter updates determined by the `Policy Improvement`. Finally, the updated trajectory stored in the policy is assigned to the group trajectory and thus to the full trajectory.

4.3 Assets and Drawbacks

Because of STOMP not employing gradient descent methods, it is possible to define cost functions, which are non-differentiable. This allows to include costs with regard to, for instance, general constraints or motor torques. Moreover, because of its stochastic approach, STOMP is able to overcome local minima, which is not the case for gradient-based methods like CHOMP. In contrast to CHOMP, STOMP needs way less iterations for finding a suitable plan, since STOMP is able to make larger steps in optimization. Overall, STOMP and CHOMP have comparable execution times [8].

In contrast to other sampling-based motion planning algorithms, STOMP outputs trajectories which do not need any further processing. While motion planners like KPIECE first find a feasible but non-optimized solution, which has to be smoothed and improved by eliminating redundant waypoints, STOMP already includes those constraints within its problem definition and cost function. Therefore, STOMP saves execution time by implicitly considering the quality of the found trajectories.

Apart from this, STOMP can easily be integrated into ROS and the implementation can be used with different robot hardware due to the use of configuration files. Since STOMP allows planning for different pre-defined planning groups, joints can be added or removed without effort.

Accordingly, STOMP has some major advantages, nevertheless there are also some drawbacks. Due to employing a stochastic approach, in general STOMP converges slower than other optimization techniques. It also needs more cost evaluations and therefore, each iteration takes more time than, for example, one iteration of CHOMP. As STOMP is a local optimization planner, it also depends on the initial trajectory for optimization and is not able to solve problems of a high difficulty level like finding paths in mazes.

In addition, STOMP is designed and implemented to only being executed once in a while. Hence, the algorithm is not able to consider dynamic obstacles, unless it is actively called consecutively to generate a new plan from scratch. Although having similar runtimes in comparison with other motion planning algorithms, the current implementation of STOMP is not fast enough for realizing effective replanning. Moreover, the addition of further joints resulting in a higher dimensional search space decelerates the planner.

Due to a fixed duration of trajectories in the implementation, the final trajectories are executed very slowly and therefore, need more actuation power to compensate gravity. Furthermore, the long trajectory duration not only leads to abnormally slow motion especially for short lengths to overcome, but also to a huge number of needless waypoints. This also contributes to a longer runtime of STOMP.

Altogether, STOMP already is an efficient planner with an interesting approach, however, it has some flaws, which lead to STOMP being slower than it could be. Because of runtime being one of the most important assets of a motion planning algorithm next to producing smooth and collision-free trajectories, it is crucial to integrate concepts such as multiresolution in time into STOMP in order to speed up the motion planning. Thus, making it possible for STOMP to being employed for tasks, which require rapid and frequent planning.

5 Continuous Replanning with STOMP

Since STOMP is a promising motion planner with many benefits, it is expedient to choose it as a basis for employing efficient motion planning. Despite its disadvantages, STOMP provides various possibilities for enhancing the planner.

In this chapter I will first present the considered approaches for solving the underlying motion planning problem and for improving the stomp motion planner. Subsequently, I will describe the actual execution of the deliberated methods and their integration into the planner. Finally, I will point out and discuss the problems, which occurred during the implementation, and their solutions.

5.1 Approach Towards Enhancing the Planning

Solving motion planning problems is computationally complex and therefore finding feasible solutions takes long. In order to employ real-time motion planning for many joints and in addition, being able to deal with dynamic obstacles, it is crucial to reduce the planning times.

The approach pursued in this thesis is to take an efficient and evaluated motion planning algorithm and improve its planning time, while keeping the quality of the solution. As a result of the speedup, additional features such as continuous replanning and planning for additional joints can be achieved.

The STOMP motion planner provides various possibilities for improvement. One important part of the STOMP algorithm is the initialization of the iterative optimization. Since STOMP converges more rapidly, if the optimization gets a good guess for an initial trajectory as an input, this can be utilized to speed up the planner by choosing the input trajectory wisely. Instead of initializing the planner with trajectories, which are newly generated by interpolating between the start and the goal configuration, knowledge of previous planning cycles can be utilized. In practice, the trajectory of the recent planning cycle can be used as an initialization. The idea is, that although the environment is dynamic, it is likely that within a short time span the vicinity of the robot does not change significantly. Hence, the previously planned trajectory is expected to resemble the solution of the current planning cycle.

The main idea to achieve a faster planning process is to reduce the number of calculations and simplify them in order to decrease the computational complexity of the planning. One way to diminish the number of waypoints is to enlarge the time intervals of the discretized trajectories. However, this changes the quality of the resulting trajectory and reduces the chance of finding a feasible trajectory. Still, reducing the number of waypoints can be accomplished by utilizing the assumption that the dynamic environment changes over time. It is reasonable to only plan precisely for the beginning of the trajectory and gradually decrease the resolution of it, as the environment might change throughout time and planned trajectories are likely to become invalid. This multiresolutional approach regarding time leads to less calculations as fewer waypoints are needed to represent a trajectory. As a result, the runtime of the motion planning process is greatly decreased, while still providing plans with a good quality compared to uniform approaches.

Reducing the runtime of the STOMP implementation makes it possible to employ frequent replanning. By means of fast and continuous planning, dynamic obstacles can be avoided and thus, the planning adapts to changing environments. Also the partially rough plan of the multiresolutional approach is refined after each planning cycle.

As the runtime of the motion planner scales with the number of dimensions of the c-space, it is necessary to first decrease the runtime before adding extra joints to the planning process. The additional joints facilitate more solutions for a given motion planning problem and hence, problems might be solved, for which no solutions were found with less joints. Moreover, the resulting trajectories become more accurate and also the abilities of the hardware are utilized to a greater extent.

The work of this thesis sets a cornerstone for further development with regard to mobile manipulation. Frequent and fast planning enables adding even more bodyparts to the planning process as well as making it possible to sequentially plan for both arms without higher delay. Also, whole-body motion planning by means of including the navigation into the planning should be possible. The multiresolutional approach can be expanded to space in terms of a number of distance fields with different sizes and resolutions. Furthermore, the robot's body parts can be arranged in groups with different resolutions and be successively activated throughout time while approaching a goal configuration by navigating and concurrently actuating the other body parts.

Within this whole concept of motion planning for mobile manipulation, this thesis focuses on the task of accelerating the planning process on the basis of the given planning algorithm, in order to facilitate the mentioned ideas for further development.

5.2 Framework Setup

As STOMP is not a standard ROS package, it is necessary to manually download the source code and integrate it into ROS. Since the publication of the STOMP algorithm is from 2011 and the code was not actively maintained, it has become outdated. Therefore, the first step is to update the code with regard to the current ROS distribution called Groovy Galapagos. That is replacing deprecated classes, functions and member variables as well as dealing with relocated or renamed classes and packages. Moreover, functions which are not available anymore have to be substituted by means of assembled or own code. As major changes were made to the whole planning environment module, updating the code to a running version was very costly. Altogether, because most of the changes only affected the interface of the STOMP motion planner, the main functionality of STOMP was preserved.

In the next step, the configuration of STOMP has to be adapted to our hardware and our given software. Because of some major updates between ROS Groovy Galapagos and its predecessor Fuerte Turtle, it is necessary to primarily update the URDF files of our robots to the current state of the middleware. Since the robot descriptions of both of our domestic service robots can be interchanged and Cosero is mainly used nowadays, its URDF is included into the STOMP configurations.

As there is already an implementation of motion planning for our robots using LBKPIECE, it is reasonable to benefit from this implementation with regard to STOMP. Therefore, the

LBKPIECE planner can in general be exchanged for the STOMP planner without major issues. Nevertheless, this only works without effort when utilizing the planning component visualizer. In order to fully integrate the STOMP motion planner, so it can be utilized by the real robot as well as the simulation of the robot, a motion planning module has to be created. The motion planning node is able to switch between different planners and planning for both endeffectors separately. It provides a service, which triggers the planning process and can be called by other nodes. Also it is possible to either sent a goal configuration in terms of joint angles or only request an endeffector pose as the motion planning node is able to employ inverse kinematics for finding a suitable goal joint configuration. Subsequently, a `planArmAction` is called, which sends the requested goal configuration to the chosen planner and gets a planned trajectory as a response. Eventually, the trajectory can be transmitted to a robot control node to be executed by the robot.

While it is possible to employ the motion planning node together with the simulation to test the STOMP motion planner, it is expedient to stick to the planning component visualizer. Although the goal configuration cannot be set by means of joint configurations, the planning component visualizer provides the possibility of setting endeffector positions within a graphical user interface. The joint configuration is then computed by means of inverse kinematics triggered by the planning component visualizer. This is more comfortable for testing and visualizing the results of the motion planner than employing the simulation and manually defining endeffector positions. Moreover, obstacles can be easily added and moved within the planning component visualizer. In order to visualize the planned trajectory, an additional trajectory visualizer is employed in Rviz.

Overall, the software environment for utilizing the stomp motion planner is set. Accordingly, the next sections focus on the changes made to STOMP in order to improve the planning as well as the runtime.

5.3 Initialization

Due to STOMP being an optimization algorithm, an initial trajectory is required as an input for finding a solution in terms of an optimized trajectory. The original implementation of STOMP takes the start and goal configuration, sets them as a start and end point in a newly created trajectory and finally, fills the intermediate points by means of B*é*zier splines. As a result a trajectory like the uniformly spaced trajectory in Fig. 4.3 is generated.

Because of the use of B*é*zier splines, the interpolation between the start and end point is not only smooth but also short. This is a huge advantage for planning scenes with no obstacles in the vicinity of the robot and its endeffector, as most of the times a nearly linear interpolation is similar to the final solution of the optimization. However, since there might be obstacles between the start and goal configuration, it is possible that these initial trajectories are not collision-free. One advantage of STOMP as well as CHOMP in contrast to other motion planning algorithms is, that the planner can be initialized even with a trajectory, which is not collision-free. Hence, it is reasonable to use the trajectories, which are created by means of B*é*zier splines, as an input for STOMP in general.

Nevertheless, STOMP is a motion planning algorithm, which performs local optimization and therefore, primarily finds locally optimum trajectories as opposed to finding global solutions. Thus, its performance changes depending on the initial trajectory supplied for optimization. Furthermore, it might be possible that no solution is found at all, due to a bad initial trajectory. As the sampled noise lies within a fixed interval, the impact of the exploration steps in each iteration are limited. Consequently, the more remote an initial trajectory is from the locally optimum trajectory, the more iterations are needed to converge to it. As a result, it is fundamental to consider using other methods for determining initial trajectories in order to reduce iterations and therefore, decrease the runtime of STOMP.

One simple approach is to reuse previously planned trajectories. It is possible to create a library of pre-planned trajectories, which can be used as an initial trajectory in similar tasks and environments. Still, it is difficult and time-consuming to determine whether a current planning scene and task match a saved situation in the library. Another possibility is to reuse recent trajectories online, which means that during a whole session of tasks previously planned trajectories are used as an input for optimization. When only using the motion planning every once in a while for different tasks in a completely different environment, reusing trajectories, which are possibly planned minutes ago, is not productive or beneficial. However, we assume that planning takes place frequently and also consecutively, as the robot might directly repeat tasks or replan while executing a task. Meanwhile, static and even dynamic obstacles in the workspace of the robot most likely stay in the same position or area. Hence, the previously planned trajectory is very probably similar or near to the locally optimum trajectory and therefore, reusing the previous plan leads to a quicker convergence of the optimizer, as the old trajectory only has to be refined.

In the improved implementation of STOMP, the reuse of previously planned trajectories is included. Overall, two different cases are considered within the planner. After starting the module and receiving the first service call, a new trajectory is created by means of the standard B*é*zier splines interpolation and fed to the optimizer. As frequent replanning is employed, the goal stays the same in each cycle and therefore, the prior trajectory is always used as an input for the next planning cycle. When receiving another service call, the new goal is compared to the goal of the former service call. If the Euclidean distance of each joint between the old and the new configuration is smaller than $5°$, it is assumed that the goals match. In this case the previous trajectory is chosen as the initial trajectory as well.

Although the approach is simple, the implementation of STOMP is too complex to simply use the former trajectory as an input. Because of all classes except for the `Stomp Planner Node` being deleted after each planning cycle, the trajectory has to be saved in the planner node. Within all other classes, the input trajectory is copied and changed in many places and furthermore, the `Covariant Trajectory Policy` keeps its own parameter representation, which is read in by the other classes in different locations. As a result, the former trajectory has to be handed over to the other classes carefully in order to be considered and furthermore, to not corrupt the optimization process. This happens when the former trajectory is not copied at each place properly.

Due to the trajectory joint parameters saved in the `Covariant Trajectory Policy` being utilized for generating new rollouts, it has to be ensured that the initial trajectory is considered on its own, that is without extra noise. Thus, the recent trajectory can be directly chosen, if it is still near optimal. In contrast, the added noise makes it possible to choose a different solution, particularly when the previous trajectory has become not collision-free in the current planning cycle. In spite of the potential of slowing down the convergence the other way around, that is the B*é*zier splines interpolation being near optimal, reusing prior trajectories is still beneficial. This is the case when an obstacle is taken out of the vicinity of the robot. Subsequently, the trajectory may be collision-free indeed, but not immediately optimal with regard to the trajectory length and actuation force. Still, this is adequate especially if this method leads to faster convergence when avoiding obstacles.

5.4 Frequent Replanning

A notably good reason for reusing previously planned trajectories as an input for the optimization process of STOMP is frequent replanning. Especially in the case of constant replanning reusing previous trajectories is beneficial, as the task as well as the goal configuration stay the same during planning. Moreover, replanning only makes sense when the whole planning process is fast enough and therefore, the decreased runtime by reusing prior plans supports this approach.

Due to dynamic obstacles in the environment, it is expedient to repeatedly perform motion planning. In order to avoid the obstacles, the environment must be sensed and considered at all times. One approach to frequent replanning is to constantly send planning request to the original implementation and get a solution trajectory as a response. However, when executing motion planning, the external modules rely on getting complete trajectories and it is not reasonable to integrate the replanning into a number of different nodes. Hence, the constant replanning should be included into the STOMP module itself.

As a result, the implementation of the planning in the `Stomp Planner Node` has to be divided into two parts. One part is to handle the service requests in the callback function and the other is to utilize the information from the service request to repeatedly plan motions. In the original implementation, when receiving a service call first the goal is checked for validity and other information is taken from the request such as the start configuration and the planning group. Then the goal and start configurations are set, collision points are generated for the robot model and the distance field of the environment is created. Finally, the initial trajectory is created and all of the created entities are passed to the optimizer. After optimization the resulting trajectory is put into the response part of the service request and is sent back to the service client.

In contrast, the new implementation distinguishes between code, which has to be executed in each planning cycle, and code, which would become redundant and therefore time-consuming when repeating the whole callback function. Since the goal configuration stays the same during one service request, it is only set once. Similarly, the collision points only have to be generated once, because the robot parts do not change during planning. Due to reusing previous trajectories, the trajectory also needs to be initialized only once. As described in the prior section, the trajectory is only changed if a new service call is received and the new goal is different from the old goal. Finally, after initializing those parts of the algorithm, the planning is started by means of a binary flag.

In order to avoid locking the callback function and as a result preventing new service calls of being received, the actual planning cycle takes place in an own thread. While the optimization happens in the thread, the callback function waits for the first planning cycle to be finished in order to fill the service response with the first optimized trajectory. In general, the solution trajectories are no longer returned as a service response, but are sent out on a topic by a ROS publisher. Therefore, every other node, which needs the planned trajectories can subscribe to this topic.

When utilizing threads, it is indispensable to assure the code being thread-safe. Consequently, mutexes are used for locking either the callback function for the setup of a new

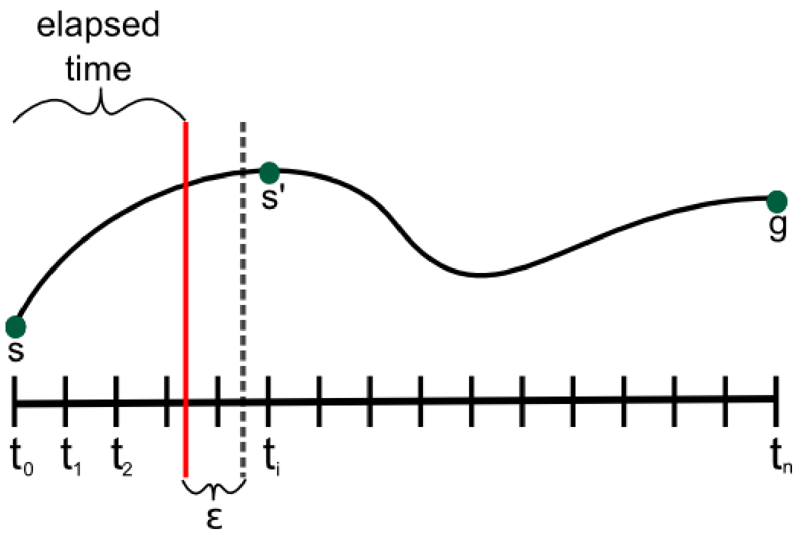


Figure 5.1: The idea of moving the start index when replanning. s and g are the start and goal configurations, s' is the new start index at time t_i and ϵ represents a time padding in the amount of the mean time needed for finishing the optimization.

planning problem, or the thread for executing the planning. If the code was not thread-safe, it could be possible that data from a previous planning cycle is used for the new planning problem and therefore corrupts the planning or the planning is even aborted due to occurring segmentation faults. Hence, in the implementation the flag ordering the spinning of the thread is unset as soon as a new call is received and as a result, the thread stops after finishing the planning cycle. Then the callback function can set up the new planning problem and subsequently, the thread can resume the planning process. As the environment has to be sensed again for each optimization, the distance field is newly created in each thread spinning. Apart from this, a `Stomp Optimizer` instance is created, which starts the optimization process, and the resulting trajectory is post-processed and published.

Overall, the replanning as such does not affect the other entities of classes like `Policy Improvement` or `Covariant Trajectory Policy`, as they are newly created every time a new optimizer instance is created. Because of those classes staying the same in each thread cycle, an idea was to save at least the policy outside of the optimizer in the stomp planner node. Nevertheless, the STOMP implementation is very complex and therefore, moving entities or functions makes the code unstable and error-prone. Thus, the instances of those classes remain to be newly constructed by the optimizer in each planning cycle.

5.4.1 Moving the Trajectory over Time

Combining frequent replanning with reusing previously planned trajectories as an input for optimization leads to two different cases. On the one hand, when replanning is executed with a different goal, the trajectory is newly created and initialized without difficulty. On the other hand, using the former trajectory as an input for planning to the same goal results in the problem of choosing a new start configuration concerning time.

The original implementation of STOMP implies the idea of a start and an end index. Both are only used within group trajectories and define the interval the optimization takes into account. As depicted in Fig. 4.4 the start and end indices do not have to be similar to the start and end point of the trajectory. Therefore, it is possible to simply move the start index to a future timestep and only optimize from this point on when replanning. Since the robot starts executing the planned trajectory right after it was published, the new starting point needs to be ahead of the current time. In order to find the new starting point, the elapsed time since the prior trajectory was published is determined and an additional value ϵ is added to this duration as a time padding in the amount of the mean time needed for finishing one planning cycle. The latter is done because the new starting point should still be ahead of the current execution time after planning. This guarantees a smooth transition between the old trajectory being in execution and the newly optimized part. Eventually, the complete duration is rounded up to a new start index as shown in Fig. 5.1.

Although the concept of the start and end index exists in the original STOMP implementation, this feature is not always taken into account within the actual implementation of the optimization. Therefore, it first has to be integrated correctly into several parts throughout the motion planner.

5.5 Multiresolution in Time

In order to realize frequent replanning rapidly and at a steady pace, it is crucial to accelerate the execution speed of the motion planning. One approach for reducing the number of calculations and thus speeding up runtime is multiresolution.

5.5.1 Concept

Whereas environments with only static obstacles are constant throughout time, dynamic environments change regularly due to dynamic obstacles as well as the motion of the robot itself. Consequently, the future state of a dynamic environment is uncertain, as obstacles might appear or move suddenly. The multiresolutional approach takes advantage of this property by implicitly taking the uncertain future into account. It is assumed that while a planned trajectory is executed the future waypoints might become invalid. Hence, it is not necessary to have a trajectory with a high resolution throughout time. In order to reach a goal configuration it is sufficient to roughly know which configurations should be taken later and already move with regard to them.

By default the duration of a trajectory is uniformly discretized into a number of timesteps. This means that after a pre-defined time interval is elapsed, a new joint configuration has to be assumed. In order to get precise and smooth movements, the intervals are chosen with a very short and constant span in the amount of milliseconds. Therefore, the positions of the joints are known accurately throughout the whole duration of the movement by means of a relatively high number of timesteps.

When utilizing multiresolution, the time intervals differ in their length with regard to the duration of the whole trajectory. With an increasing offset to the starting time of the trajectory, the resolution of the time intervals decreases. As a result, the lengths of the time intervals increase by means of a pre-defined function determining the growth of duration between timesteps. Accordingly, fewer timesteps are needed for covering the whole period of the plan.

Due to a smaller number of timesteps, less calculations have to be conducted and moreover, memory is needed to a lesser extent. Overall, the multiresolutional approach has a computational advantage compared to the uniformly discretization of the trajectory length. However, the planned multiresolutional trajectory is not as precise and smooth as the uniformly spaced one and therefore, it would be difficult to execute the trajectory properly. Nevertheless, by frequent replanning, the multiresolutional trajectory will be refined gradually. Not only does this yield a trajectory which has a comparable quality with regard to the trajectory with uniformly discretized timesteps, but also using a multiresolutional trajectory as an input for replanning benefits in an even faster computation.

This advantage holds for the case of an occurring or moving dynamic obstacle as well as the case of lacking dynamic obstacles in the vicinity of the robot. Since it is unknown whether or not an obstacle will appear, it is inevitable to employ frequent replanning in order to consider dynamic obstacles. Due to fewer timesteps and therefore also fewer calculations, the multiresolutional spaced trajectory will be optimized faster than the uniformly discretized equivalent in general. This means, that in the case of no obstacle appearing at all, the multiresolutional approach is still faster. Concerning the case of dynamic obstacles interfering with the workspace of the planned path, fewer timesteps become invalid for the multiresolutional trajectory. Thus, a smaller amount of timesteps has to be changed and optimized further.

Apart from multiresolution in time, it is also possible to employ local multiresolution in space in order to speed up collision checking. Regarding STOMP one possibility would be to create a number of distance fields with different sizes and resolutions centered at the endeffector. However, there are no real benefits to this idea, as the global distance field itself is already efficient enough for collision checking and further voxelgrids would complicate the calculations. Nevertheless, it would be reasonable to have multiple grids with different sizes and resolutions for specific body parts such as the basis of the robot in contrast to the arm. Since the basis is way larger than the arm with the endeffector, it is sufficient to use a grid with lower resolution for the navigation. However, as the work of this thesis focuses on motion planning for the upper body of a robot, that is excluding the basis, it is reasonable to stick with one distance field with elaborate parameters and focus on multiresolution in time.

5.5.2 Multiresolutional Trajectory Representation

Since the original implementation of STOMP only utilizes trajectories with a uniform discretization of time, a new multiresolutional representation has to be added to the given code. Moreover, the new trajectory representation has to be integrated into the other classes, which deal with the optimization of the trajectories.

As described in section 4.2.2 new trajectories are constructed given a trajectory duration, trajectory discretization, and a number of waypoints. All of these variables are fixed and pre-defined in a configuration file for the stomp motion planner. Although only two of the three variables have to be set to compute the value for the third variable and two constructors are implemented exactly for this purpose, the original implementation assumes that all three of the variables are correctly set within the configuration file. Otherwise, the optimization fails and diverges or the whole planner crashes. While it is expedient to pre-define a fixed discretization for the trajectories, it is disadvantageous to also use a pre-defined constant value as the duration of each trajectory being planned by the stomp motion planner. As the trajectory duration, which is already set in the configuration file from downloading the STOMP implementation, is $5\,s$, especially short movements are not only executed very

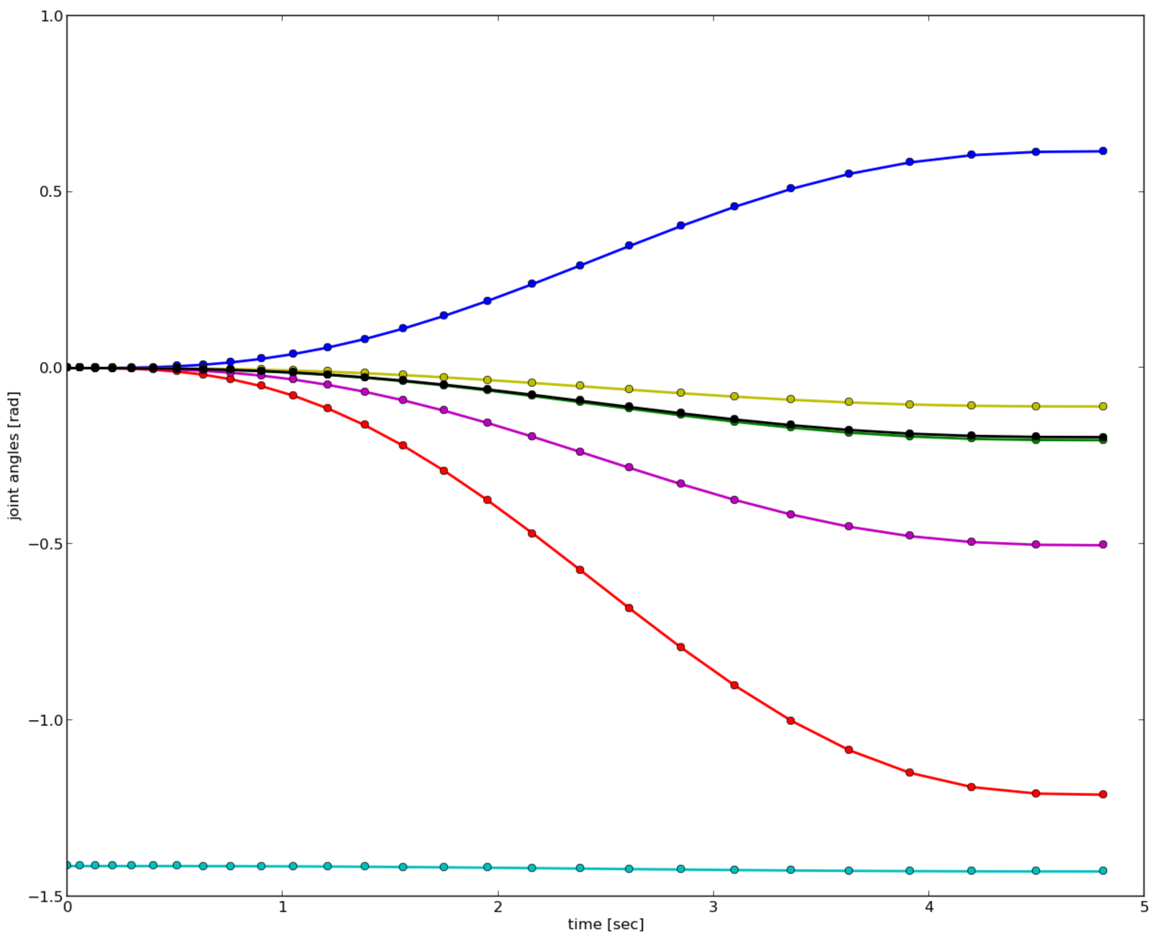


Figure 5.2: A set of initial trajectories with multiresolutional spacing for the joints of interest, that is the 7 joints in the robot arm.

slowly but also the optimization process unnecessarily gets complex. In contrast, when decreasing the fixed trajectory duration, longer and more complex movements will be executed too fast and therefore, safety is not guaranteed and the robot hardware might be damaged. Consequently, it is necessary to dynamically determine the trajectory duration for a given motion planning problem.

Since the start and goal configurations are given, the Euclidean distance between the configurations can be calculated for each joint. In addition, the maximum speed of the built in Dynamixel actuators without load is specified in the manual, which is $64rpm = 6.7rad/s$ for the RX-64 and $91rpm = 9.53rad/s$ for the EX-106+ [14]. By means of those values and the weight of the construction parts an educated guess of $0.3rad/s$ regarding the average velocity of the actuators can be made. As a result, the duration of a new trajectory can be calculated by

$$d = \frac{max_{j\in J}(|x_{j,N} - x_{j,0}|)}{v} + \epsilon \tag{5.1}$$

where J is the number of joints, $x_{j,0}$ and $x_{j,N}$ are the start and goal configurations for a specific joint and v is the average velocity of the actuators. Due to the fact, that the goal configuration cannot be reached by means of a linear interpolation when obstacles are blocking the way, an extra time padding ϵ is added to the calculated duration.

In order to finally compute the number of waypoints, a trajectory discretization has to be defined in the configuration file. In case of uniformly spaced trajectories, the resolution of the trajectory is defined by one value. In contrast, the multiresolutional trajectory needs a minimum trajectory discretization as well as a function for determining the growth of the time intervals. Although this function can be chosen arbitrarily, the length of the time intervals should not become too large. Because then the number of waypoints might get too low to find feasible plans and also collision avoidance might fail. The chosen function for the multiresolution in terms of the balance between reducing the computational complexity and still having enough points for efficient planning is a linear function defined by

$$f(x) = 0.01x + r \tag{5.2}$$

with r being the minimum resolution for discretization. In general a good value for discretization with regard to time is $0.05\,s$ and therefore, the value can also be used as a minimum resolution for the multiresolutional trajectory.

After determining the duration of the trajectory as well as the highest resolution and the growth function $f(x)$, the number of points can be computed. Accordingly, the time intervals calculated given the growth function are summed up and saved in a vector until the sum of all the time intervals exceeds the previously determined duration of the trajectory. Meanwhile, the number of waypoints is raised with each addition and finally equals the number of time intervals plus one extra point referring to the goal configuration. Since the sum of all intervals usually does not equal the duration of the trajectory, the duration is adjusted to the sum.

Eventually, the new trajectory can be created by means of duration, number of points and the vector including the determined intervals, with the latter being saved within the multiresolutional trajectory structure. Similar to initializing the uniformly spaced trajectory by means of B*é*zier splines (see Fig. 4.3), the multiresolutional trajectory has to be initialized. For filling the intermediate points of the multiresolutional trajectory, B*é*zier splines are used as well except for now calculating the spline coefficients and the joint positions at the timesteps with regard to the multiresolutional spacing. Like the uniformly spaced trajectory, the trajectory with multiresolutional intervals is short and smooth (see Fig. 5.2).

5.5.3 Replanning with Multiresolution

When employing frequent replanning, the starting point of a reused trajectory has to be changed, as stated in section 5.4.1. Accordingly, the new start index for optimization is easily determined with regard to the elapsed time since the last planning cycle finished (cf. Fig. 5.1). However, when using multiresolutional trajectories, adapting a trajectory for replanning gets more complicated.

Due to not only using an old trajectory as an input for planning, but also refining the trajectory, simply moving the start index becomes insufficient. If only the start index was changed, it would be pretty difficult to keep track of the resolutions of each timestep. Moreover, the timing of the intervals and the whole trajectory duration would become impossible without changing either the latter or changing the duration of the last time interval.

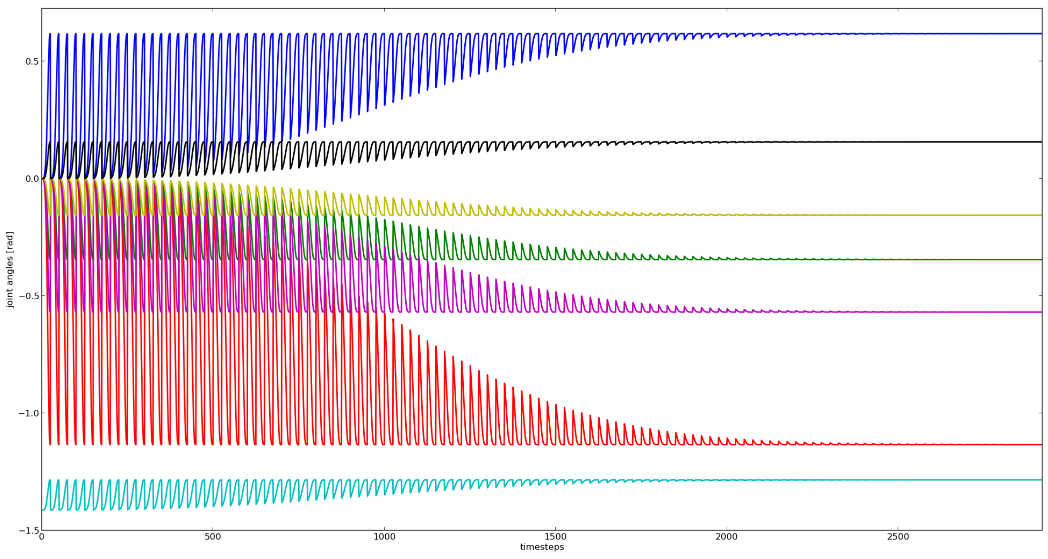


Figure 5.3: Trajectories for each joint moved by one timestep after each planning cycle.

Consequently, it is wise to move the whole trajectory instead of moving the start index. As a result, the resolution as well as the trajectory duration is restored. Nevertheless, the number of points also stays the same, which has to be taken into account, when moving the trajectory. Overall, it is important to find a good strategy for shifting the trajectory. Two different possibilities are shown in Fig. 5.4. In both ways the new starting point is determined as in Fig. 5.1 and subsequently, the new starting point is moved to the beginning of the trajectory. The following waypoints then have to be moved successively. In the left column the method of just moving the waypoints with regard to their index and without taking the time intervals into account is shown. As a result of this method, the gradient of the trajectory is completely changed and therefore, the optimization process will get corrupted. Hence, the multiresolution has to be considered when moving the trajectory. In the right column the old trajectory is sampled from the new starting point on regarding the differently sized time intervals, which leads to the gradient being mostly retained. However, refining the trajectory during the moving process leads to smoothing and truncating of the trajectory as depicted in the figure. Altogether, both approaches are not ideal, still, the second approach does not corrupt the optimization process and is therefore favored.

Since the idea in the right column of Fig. 5.4 is implemented, the effect of moving the trajectory can be shown in terms of an actual trajectory set (cf. Fig. 5.3). In the plot trajectories for each joint of the robot arm are depicted while being moved about one timestep after each planning cycle. Thus, the trajectories converge towards the goal configuration. In the real application it is not likely that the trajectory will always be moved just by one timestep, but also multiple timesteps can be skipped.

As the duration and the number of timesteps are consistent, the movement speed tends to slow down. Nevertheless, as the rear waypoints converge to the goal configuration, the executing robot control will stop as soon as the robot is in the close vicinitiy of the goal configuration. Because of applying the optimization to the group trajectory instead of the full trajectory, the extra waypoints in front of the starting point also have to be considered. Consequently, the waypoints are sampled in terms of the minimum trajectory resolution

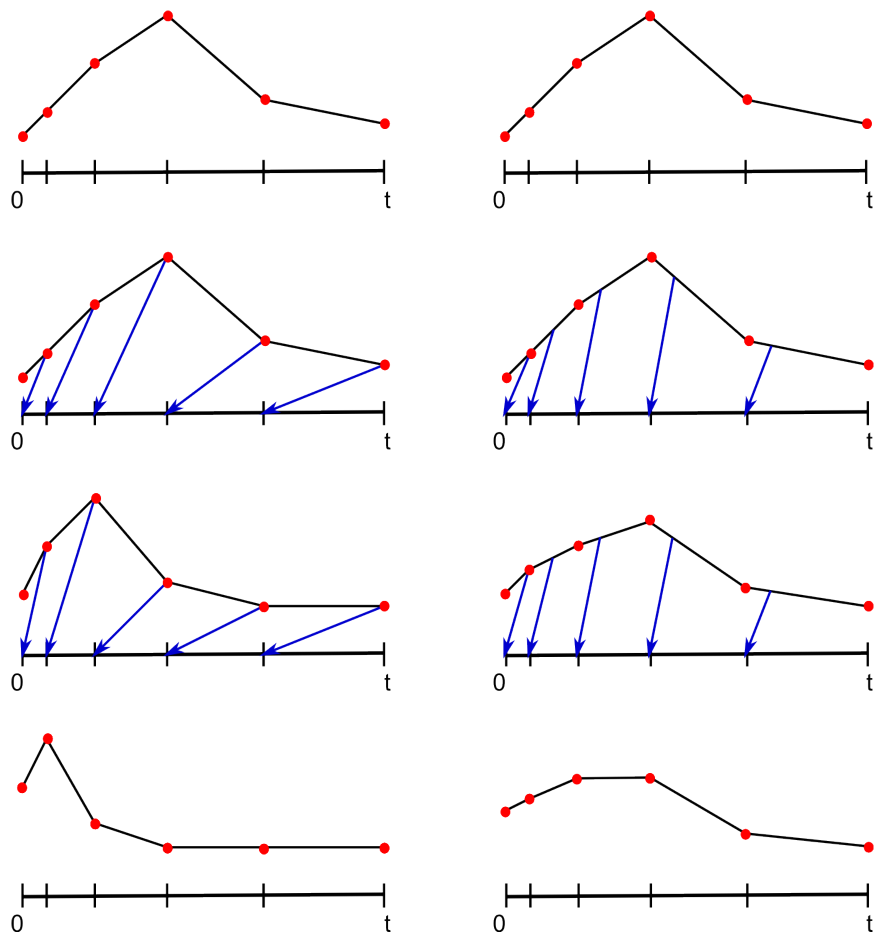


Figure 5.4: Two possibilities for moving multiresolutional trajectories in time. The left column shows a naive approach where the waypoints are directly moved. This leads to the problem that the trajectory is shrinked. In contrast the right column depicts an interpolation approach for moving.

from the starting point on in the backwards direction (cf. Fig. 5.5). That allows to correctly compute the velocity and acceleration of the trajectory when reusing it as an input.

5.5.4 Control Costs

The control costs make use of the velocity and acceleration of the trajectory for which the costs are calculated. Moreover, the obstacle costs use the magnitude of the workspace velocity of the collision spheres of the robot to ensure that the robot does not move through high-cost areas rapidly in order to lower the costs. Therefore, both, velocities and accelerations, have to be computed.

In the original implementation three filters, namely the differentiation rules, are predefined for the purpose of calculating the first, second and also third order of derivatives. Those filters are then used to create the so-called differentiation matrices regarding each order of derivatives. In the case of the acceleration the differentiation matrix correlates with the matrix **A** in Eqn. 4.2. The differentiation matrices are then used to define the matrix

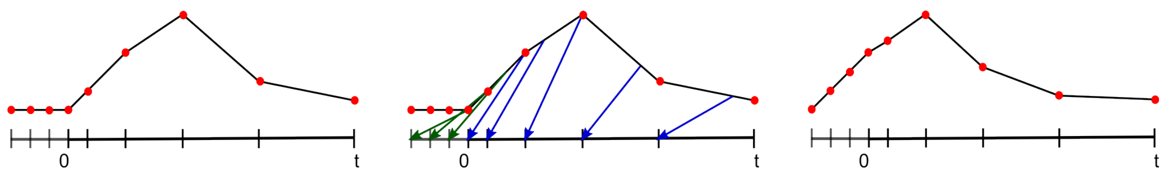


Figure 5.5: Moving a multiresolutional group trajectory in time for replanning and determining the correct accelerations in particular. The group trajectory is moved for two timesteps.

Table 5.1: Coefficients for centered approximations at a grid point with variables $M = 3$, $N = 6$, $x_0 = 0$ and $\alpha_\nu = \{0, 1, -1, 2, -2, 3, -3\}$ utilized in the original implementation of STOMP.

Order of		x-coordinates at nodes						
derivatives	accuracy	-3	-2	-1	0	1	2	3
1	4		$\frac{1}{12}$	$-\frac{2}{3}$	0	$\frac{2}{3}$	$-\frac{1}{12}$	
2	2			1	-2	1		
	4		$-\frac{1}{12}$	$\frac{4}{3}$	$-\frac{5}{2}$	$\frac{4}{3}$	$-\frac{1}{12}$	
3	4	$\frac{1}{8}$	-1	$\frac{13}{8}$	0	$-\frac{13}{8}$	1	$-\frac{1}{8}$

R, which represents the control costs and is utilized in the definition of the underlying optimization problem of STOMP shown in Eqn. 4.1. **R** is created as in Eqn. 4.4 by multiplying the transpose of **A** with **A**. Also the inverse control costs $\mathbf{R}^{-1}$ are calculated and used for sampling noise amongst others. Both are displayed in plots shown in Fig. 4.1.

In order to get the three filters, an algorithm for finite differencing by Fornberg is employed [6]. The algorithm utilizes two simple recursion relations for determining the weight for calculating any order of derivatives and furthermore, they can be approximated to any order of accuracy. In contrast to previously described algorithms, the algorithm of Fornberg is not limited to a uniform discretization of grids. As the algorithm is fast due to only using four arithmetic operations, it allows to calculate the finite differencing weights for dynamically changing grids. Of course it can be also used to pre-calculate tables of weights. Moreover, it is possible to not only determine centered filters for approximating the derivatives, but also one-sided approximations can be computed.

The most interesting of the given tables includes filters for centered approximations within a uniformly spaced grid (cf. Tab. 5.1). When looking at the matrix **A** again, it can be noticed that **A** includes the finite differencing weights for the second order of derivatives with an order of accuracy of two. In the actual implementation weights with an order of accuracy of four are employed. Since the weights refer to a uniform spacing of $\Delta x = 1$, the weights are divided by $(\Delta x)^m$ when creating the differentiation matrix **A** with Δx being the chosen uniform trajectory discretization and m being the order of derivative.

Overall, the algorithm for calculating the weights is described in Alg. 2. As an input the algorithm gets $M \geq 0$ being the order of the highest derivative to be approximated, x_0 being the point, which should be differentiated, and $N \geq 0$ being the size of the resulting filter with regard to a set of $N+1$ grid points $\alpha_\nu = \{\alpha_0, \ldots, \alpha_N\}$. As a result, the algorithm outputs matrices $\boldsymbol{\delta}^m$ of the dimensions $N+1 \times N+1$ including the weights for each order of accuracy with the maximum accuracy being $N+1$.

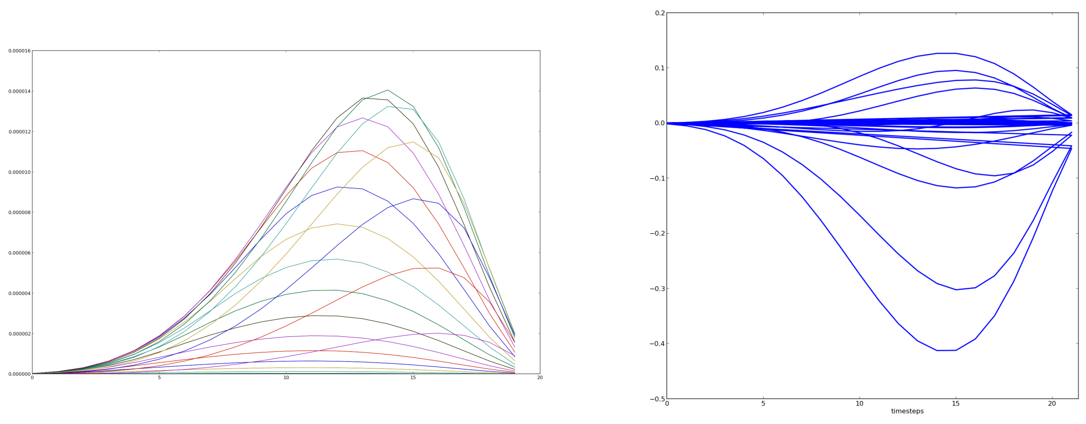
```
Input: $M, N, x_0, \alpha_0, \alpha_1, \ldots, \alpha_N$
$\delta^0_{0,0} = 1$
$c_1 = 1$
for $n = 1$ to $N$ do
    $c_2 = 1$
    for $\nu = 0$ to $n-1$ do
        $c_3 = \alpha_n - \alpha_\nu$
        $c_2 = c_2 * c_3$
        if $n \leq M$ then
            $\delta^n_{n-1,\nu} = 0$
        for $m = 0$ to $min(n, M)$ do
            $\delta^m_{n,\nu} = \frac{((\alpha_n - x_0)\delta^m_{n-1,\nu} - m\delta^{m-1}_{n-1,\nu})}{c_3}$
    for $m = 0$ to $min(n, M)$ do
        $\delta^m_{n,n} = \frac{c_1}{c_2}(m\delta^{m-1}_{n-1,n-1} - (\alpha_{n-1} - x_0)\delta^m_{n-1,n-1})$
    $c_1 = c_2$
```

Algorithm 2: The finite differencing algorithm by Fornberg. If $m = 0$ the term $zero * (undefined\ number)$ is assumed to be zero.

Since the derivatives for calculating the velocity and acceleration are also needed when applying the multiresolutional approach, the finite differencing filters have to computed with regard to the multiresolutional waypoint spacing. Hence, the Fornberg algorithm is implemented within the new STOMP implementation.

Unlike the uniform approach, it is not sufficient to only calculate one filter for the whole trajectory and subsequently, divide the filter by the resolution of each point. In order to have correct filters for each point in the trajectory, the finite differencing algorithm also has to be applied for each point separately. This means, that for every waypoint the x-coordinates $\alpha_\nu = \{\alpha_0, \ldots, \alpha_N\}$ have to be defined. This is done by utilizing the vector in which the resolutions of each point is saved. In case of the start and end point, the preceding and succeeding points in the group trajectory have the same resolution as the start and end point, respectively. Otherwise, the multiresolutional time intervals of the neighboring points are added and accordingly substracted from another regarding whether they are chronologically before or after the currently deviated point. For example, the α_ν for the starting point with $0.05s$ being the highest resolution, Eqn. 5.2 being the growth function and a filter size of six is $\alpha_\nu = \{0, 0.05, -0.05, 0.11, -0.1, 0.18, -0.15\}$.

After the α_ν are created for each timestep, the finite difference algorithm by Fornberg is used to get three matrices, one for each order of derivative, for every point in the trajectory. Finally the fourth order of accuracy of each of the orders of derivative is used to build the multiresolutional differentiation matrix $\mathbf{A}$. As the resolution is already considered while determining the derivative filters, unlike the uniform filters, it is not necessary to



(a) Multiresolutional inverse control cost matrix (b) Sampled noise

Figure 5.6: (a) Each curve represents a column and a row of the inverse control cost matrix $\mathbf{R}^{-1}$ regarding multiresolutional finite differentiation. (b) 20 random noise samples $\boldsymbol{\epsilon} = \mathcal{N}(0, \mathbf{R}^{-1})$.

scale the multiresolutional filters by additionally dividing them by a Δx. Referring to the previously described example, the differentiation matrix of the second order of derivative for a trajectory with 10 waypoints and two being the order of accuracy is

$$\mathbf{A} = \begin{bmatrix} -10 & 0 & 0 & & 0 & 0 & 0 \\ 1 & -10.9091 & 0 & \dots & 0 & 0 & 0 \\ 10 & 3.33333 & -8.97436 & & 0 & 0 & 0 \\ 0 & 7.57576 & 2.38095 & & 0 & 0 & 0 \\ & \vdots & & \ddots & & \vdots & \\ 0 & 0 & 0 & & 0.909091 & -4.74308 & 0 \\ 0 & 0 & 0 & & 4.329 & 0.757576 & -4.33333 \\ 0 & 0 & 0 & \dots & 0 & 3.98551 & 0.641026 \\ 0 & 0 & 0 & & 0 & 0 & 3.69231 \end{bmatrix}. \tag{5.3}$$

The multiresolutional matrix $\mathbf{A}$ is used in the same way as the uniform equivalent in order to build the control cost matrix $\mathbf{R}$ and subsequently, the inverse control costs $\mathbf{R}^{-1}$. Due to the multiresolution, the matrix $\mathbf{R}^{-1}$ is not symmetric anymore while still being positive definite. In Fig. 5.6 the inverse cost matrix $\mathbf{R}^{-1}$ is depicted as well as the sampled noise by means of this matrix. The noise gets higher towards the end of the trajectory, which allows stronger deviation with regard to longer time intervals. Unfortunately the noise slightly diverges from the goal configuration in contrast to the uniformly sampled noise. Nevertheless, this does not lead to bigger problems when optimizing, since the divergence is very small.

In addition to using the inverse control costs for sampling noise, the control cost matrix is of course used to calculate the control costs of the rollout trajectories in terms of accelerations. Moreover, the derivatives are used for calculating the magnitude of the workspace velocity of the collision spheres. For this reason the finite differentiation filters of the first and second order of derivatives are applied to the positions of the collision spheres of each robot body part throughout the trajectory and subsequently, they are multiplied by the time intervals and accordingly squared time intervals at this point of the trajectory.

5.6 Adding Joints

Motion planning is mostly employed for robot arms and endeffectors in order to solve manipulation problems. However, it is expedient to include additional joints into the planning process, as more degrees of freedom might lead to better solutions or even finding a solution at all, when planning only with the arm and endeffector fails. Since adding extra joints to the motion planning process yields to an additional dimension in c-space, the optimization slows down. Thus, the planning should be fast in the first place in order to take additional joints into account.

The implementation of STOMP utilizes configuration files regarding the body composition of the robot. In those configuration files it can be specified which joints of the robot should be used for planning by defining a planning group as well as the base and tip links for the kinematic chain of the robot. Moreover, it can be defined which of the joints should be tested for collisions against each other. As it is physically impossible for some of the joints to collide with each other, the testing for those joints can be disabled. Accordingly, the number of collision checks greatly decreases, which leads to a speed up of the planning process.

Basically, adding supplementary joints to the planning with regard to the stomp motion planner is not difficult. However, appending joints might lead to side effects within the planner. Due to the torso joints of our robots not only being in the kinematic chain for the right arm and the head, but also in the second kinematic chain including the left arm, the `Stomp Robot Model` samples collision points from the whole torso including both arms and the head. Still, the left arm is included as an obstacle in the distance field, as it does not belong to the planning group. Consequently, when trying to plan for the whole upper body, the collision points of the left arm instantly collide with its obstacle representation in the distance field. The collision spheres of the upper body as well as the vectors indicating the distance from the collision points to the obstacles in the distance field are depicted in Fig. 5.7. Those alleged collisions lead to overshootings of the supposedly optimized trajectory, which worsen with each new planning cycle.

In summary, the implementation of the planner does not consider the existence of a torso hinge joint as it is integrated into the body composition of our robots. The torso yaw joint has the feature to move the whole upper body in its entirety and uncoupled from the rest of the body. Accordingly, both arms are always moved as a result of the actuation of the torso. In contrast to this characteristic of our robots build, the implementation of STOMP assumes, that actuating a joint only leads to the motion of one kinematic chain attached to the joint. Therefore, the planner characterizes the left arm as a fixed obstacle, while still including it into the collision point sampling.

In order to prevent this contradictory behavior, it is necessary to exclude the left arm from being registered within the distance field. Finally, the motion planning can be executed

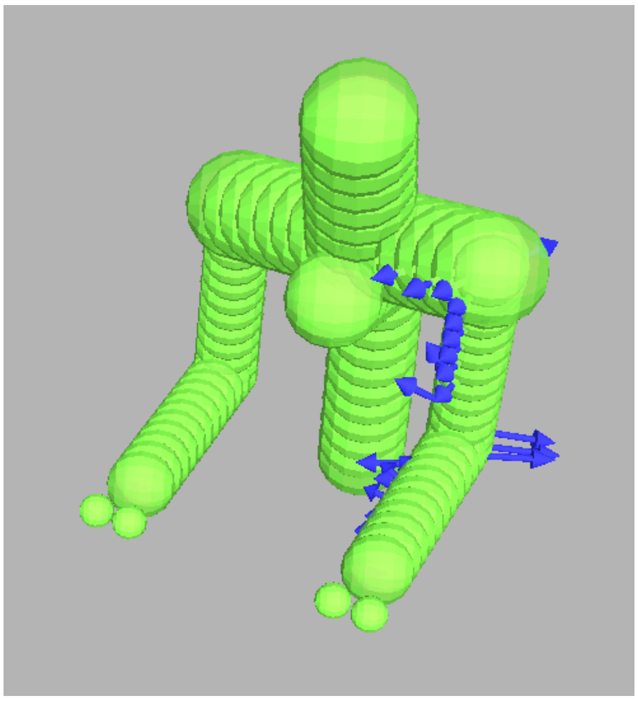


Figure 5.7: The points of the upper body of the robot used for collision checking represented by spheres. The blue vectors indicate collisions with the distance field entries of the left arm. The vectors point away from the distance field.

correctly again.

Another special characteristic of the body composition of our robots is the torso lift. Unlike all other joints in the robot being revolute joints, the torso lift is a prismatic joint. It can be used in order to move the upper body of the robot up or down regarding a pole link in order to, for instance, grasp objects from the ground or from high shelves. Similarly to the torso yaw joint, it can enable additional possibilities for avoiding collisions. However, actuating the torso lift needs a lot of propulsion and therefore, should be avoided if not really necessary. It is possible to weight the torso lift joint, so the solution trajectories, which employ the lift have higher costs than the trajectories only moving the arm and the torso yaw. Nevertheless, it is not feasible to completely forbid moving the lift in certain situations. Furthermore, the robot control discretely manages the use of the torso lift and therefore, the planner does not have the full impact on the movement of the torso lift. Still, if it is decided to include the torso lift into the motion planning, it can easily be added.

5.7 Adjustments and Configurations

The cost function of the STOMP algorithm consists of three different costs, namely the obstacle cost, the constraint cost and the torque cost. Since the torque costs for our robots cannot easily be calculated, another cost has to be employed in order to ensure that the needed actuation power is low. Moreover, the optimized trajectory should be as short as possible, while still avoiding obstacles. Therefore, an Euclidean distance cost is added to the cost function in Eqn. 4.13. The one-dimensional Euclidean distance between each waypoint is calculated and summed up to the Euclidean distance costs. Subsequently, the whole costs are added to the total costs of each waypoint. As a result, the trajectories are kept short and the trajectory quickly bounce back, after an obstacle left the vicinity of the trajectory.

Overall, the implementation of STOMP provides various variables, which can mostly be defined in configuration files. Regarding the original implementation of the STOMP algorithm the discretization, duration and number of time steps of the trajectory can be set, whereas the new implementation only allows to define a minimum discretization, which represents the highest resolution of the trajectory and is set to $0.05s$. Apart from this, weights for each part of the cost function and weights regarding smoothness can be set and therefore enabled or disabled. Altogether, most of the weights were adopted from the original implementation. In addition, the Euclidean distance cost weight was adjusted with regard to the resulting level of the other costs in order to extenuate the influence of the Euclidean distances in the overall costs.

Moreover, the values for the noise and noise decay, which is applied after each iteration, can be chosen. This is also done regarding the outcome of the noise sampling and with respect to the observed bouncing of the solution trajectory after a new planning cycle. Also, the overall number of rollouts and the number of rollouts bein reused from the recent iteration can be defined. The values are taken from the original STOMP implementation with 10 overall rollouts and 5 reused rollouts. Another set of parameters are the sizes and the resolution of the underlying distance field. Since the robot's workspace and the sizes of the obstacles are limited, the scale of the distance field is set to $2m$ in each direction with a drift of the origin in front of the robot. The resolution of the distance field is defined with $0.015m$, which is supported by experiments described in chapter 6.

Two very important parameters are the overall number of maximum iterations and the number of maximum iterations after a collision-free solution is found. The former is used as a terminating condition in the case no collision-free path could be found and therefore, the optimizations should be aborted. In contrast, the latter is employed to assure that the optimization has enough time to converge. Since optimization especially for more than just the arm joints takes its time and still optimization should not take more iterations than needed, it is wise to choose a relatively low value for the number of maximum iterations after a collision-free solution is found. The same conditions apply to the overall number of maximum iterations, though it should be a lot higher than the other value. Overall, the values chosen for both variables are 20 iterations and 500 iterations, respectively, as both values lead to good outcomes.

5.8 Discussion

The STOMP implementation with continuous replanning and multiresolution provides, similarly to the original STOMP implementation, feasible, collision-free, and smooth trajectories as an output. Overall, the planner solves basic motion planning problems, which occur in the given dynamic household environment, but it is not able to solve problems of a high difficulty level such as finding paths in mazes.

The planning is not only limited to the joints of the arm and the endeffector, but also can be extended to additional joints as in case of our robots are the torso yaw joint and the torso elevator joint. Since the elevator joint has been successfully integrated into planning, it is possible to not only execute planning for revolute joints, but also for prismatic joints.

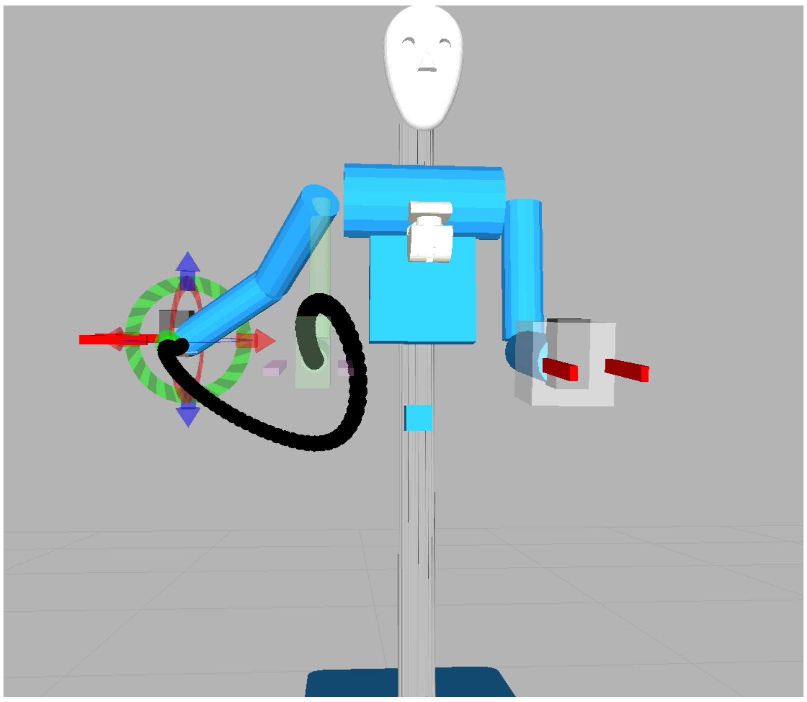


Figure 5.8: An optimized trajectory, which is overshooting at the beginning of the trajectory due to corrupted costs. This behavior occurred often when altering the STOMP code, as the implementation is sensible to changes.

However, it is too complicated to include the omnidirectional basis into the STOMP motion planner, so the already implemented navigation in terms of global and local planners has to be connected with the motion planning in order to employ whole-body motion planning. Sequential planning for both arms of the robot can be easily enabled by first planning for the kinematic chain of one of the arms, fix the result and subsequently plan for the kinematic chain of the other arm with regard to the fixed plan. Because of our robots having two different kinematic chains for each arm, it is not simple to employ motion planning for both arms simultaneously. Also, bi-manual tasks require both arms to be coordinated and thus, it is a challenging problem on its own.

In contrast to other motion planning methods, STOMP is not able to utilize redundant joints of the robot, which is referring to our robots the elbow pitch joint. This is due to the fact, that the input for the planner is defined in joint space and the goal configuration for each joint is a given constant angle. As a result, when moving obstacles in the way of the redundant elbow joint, the planner will return no plan and indicate that no collision-free trajectory could be found, as the obstacle was put in the given goal configuration.

Generally, the planner is limited to the given goal configuration. The decision whether an additional joint is utilized for planning and the determination of the goal configuration is not made within the improved implementation of STOMP. Although a motion planning module utilizing STOMP was implemented, it is only able to translate endeffector positions to joint configurations by means of inverse kinematics. Thus, more criteria have to be considered to enhance the input for the STOMP motion planner.

As already stated, the improved implementation of STOMP facilitates frequent replanning. While a planned trajectory is executed, the remaining trajectory is optimized further with the next to current waypoint as a new start configuration. Hence, dynamic obstacles appearing in the vicinity of the trajectory currently being executed can be avoided. When applying the motion planning on the real robot, the collision detection has to rely on the obtained sensor data as well as the sensor data processing. Obstacles might not be detected because they can be covered by the robots own body parts. Also, the processing of the sensor data as well as the response time of the robot control and the servos lead to delays regardless of the executed motion planning.

Unlike the original implementation of STOMP, the duration of the solution trajectory is now determined for each new planning request. Choosing a fixed trajectory duration might have the benefit of the trajectory length being always long enough for a given task. Moreover, the trajectories can be easily compared to each other when having the same duration. Still, adapting the trajectory duration to the given motion planning problem with regard to the start and the goal configuration, enables the trajectory to being executed in comparable velocities. Besides, the propulsive force of the robot's servos is utilized to a greater extent. When an obstacle approaches the trajectory and the path has to be changed, the duration still stays the same, which leads to a speedup in the case of avoiding an obstacle. An additional value is added to the determined trajectory duration to take this into account. Nevertheless, the obstacle avoiding trajectories might be executed slightly too quick, though staying within the velocity limits.

The trajectory initializing the optimization process can be chosen arbitrarily. One possibility is to create a library with trajectories for different occasions and scenarios. Another possibility is to take the most recent trajectory, because especially when replanning the former trajectory is very likely to resemble a newly optimized trajectory. Similarly, other approaches for an initial trajectory can be chosen. Due to determining individual trajectory durations for each planning problem, the duration of the input trajectory has to match the calculated trajectory length or the durations have to be adapted in either way.

Altogether, STOMP is an efficient algorithm, which in theory can be easily improved. However, the implementation of the algorithm is very sensitive with respect to slight changes. Only small changes are enough to let the trajectory diverge consistently as depicted in Fig. 5.8. Therefore, it is important to not only understand the algorithm, but also to thoroughly comprehend the structure of the implementation.

Overall, it is possible to apply the original STOMP as well as the improved version of STOMP to other robot hardware. When using an own robot model, the body composition and characteristics have to be considered in order to configure the planner and the robot model accurately and to prevent possible side effects on the optimization. Information about the kinematic chains, joints and links have to be present to build a suitable robot model. Besides to our robot, the STOMP motion planner was knowingly employed on the PR2 by Willow Garage in the original publication of STOMP [8].

The motion planning approach presented in this thesis focuses on domestic service applications. Accordingly, enhancing the given motion planner is based on the requirements of household environments and tasks. The emphasis in this setting lies on feasible, collision-free, and fast planning in unknown and dynamic environments. In contrast, other environments such as industrial settings depend on precise motions of the robots, which are replicated very often. Moreover, in those scenarios usually no unknown or dynamic obstacles appear. Consequently, rapid planning is not the priority unlike precise planning. Hence, the STOMP algorithm is not serviceable in those environments. Still, the algorithm proves to be beneficial in the underlying scenario of this thesis.

6 Experiments and Evaluation

In order to rate the implemented approach, experiments have to be made and their results have to be evaluated. Hence, it is necessary to establish criteria for measuring the success or failure as well as the quality of the implementation. Data has to be obtained and then processed by means of different assessments, which will be outlined below.

For evaluation three different motion planners are compared to each other, being the given KPIECE implementation, the original STOMP implementation as well as the new STOMP implementation developed for this thesis. The three motion planners are checked against each other, in order to determine which planner is superior, inferior or equal to the others in various parts of the planning. Of course, the focus of the evaluation lies on the modified version of STOMP presented in this thesis and whether it is reasonable to employ it or even develop it further.

In this chapter, the executed experiments as well as criteria for evaluation are described, followed by the results and a discussion of the findings.

6.1 Criteria

The modified motion planner on basis of STOMP is evaluated with regard to various criteria. Since decreasing the runtime of the original STOMP implementation is one of the main goals of the work in this thesis, the improved STOMP implementation is evaluated concerning its runtime in particular.

The runtimes of the following features are measured and compared:

- *Overall runtime.* For all three motion planners the overall computation times are measured and analyzed. The time for the filtering process following the execution of KPIECE is left out here and added later.
- *Optimization process.* The duration just for the optimization process of the original and the modified STOMP is determined.
- *Creation of distance field.* The duration for the creation of the distance field in every planning cycle is determined to see which part of the planning takes how long.
- *Finite differencing.* Since the finite differencing algorithm has to be executed for the multiresolutional approach, it is important to know its performance for a different number of waypoints.
- *Multiresolutional versus uniform.* The improved STOMP implementation is executed with a multiresolutional growth function as well as with a uniform growth function.
- *Additional joints.* To test the scaling of all three planners, extra joints are added consecutively and the slowdown of the planning is measured.

- *Initialization.* The motion planning is executed with and without using the recent trajectory as an initial trajectory for the optimization process. Simultaneously, the successful operation of the frequent replanning can be evaluated.

All of the above mentioned experiments are made without any obstacles and with a pole blocking the straight way from the start to the goal configuration.

In addition to the runtime tests, other criteria are applied to evaluate the approach. Those criteria focus on the quality of the planned trajectories and the capability of the planners.

- *Duration of the trajectories.* Since the multiresolutional STOMP determines the duration of the trajectory for each planning problem, it is interesting to look at the values. However, the trajectory duration for the original STOMP is fixed and KPIECE itself provides no information about the duration of the trajectories whatsoever.
- *Number of waypoints.* The number of waypoints not only scale the number of calculations, but also give insight about the duration of the trajectories especially concerning KPIECE.
- *Success.* Besides having a short runtime, planning feasible plans is one of the most important goals of motion planning. Therefore, the three motion planners are tested with regard to their success rate.
- *Number of iterations.* Both STOMP versions are tested regarding the iterations needed for converging. This also is connected to the resulting overall runtime and apart from this indicates a good intialization of the planners.

Overall, the experiments cover both quantitative and qualitative evaluation. The comparison between the three different planners and furthermore, the different versions of the improved STOMP implementation, notably is interesting, since the goal is to enhance motion planning.

6.2 Data Acquisition

Overall, the experiments are all executed in simulation within the ROS environment. The measurements of the robot and the initial joint configurations are obtained through the given URDF files of the robot. In order to define goal configurations or place obstacles, Rviz and the planning component visualizer plugin are utilized.

The planning experiments are executed on a desktop computer with an Intel Core i7 940 quadcore CPU with $2.93GHz$ and $24GB$ RAM. On the computer the distributions Ubuntu 12.04 LTS Precise Pangolin as well as ROS Groovy Galapagos are installed. Another experiment regarding the distance field classes is executed on a DELL Precision M4400 Laptop with an Intel Core 2 Duo T9600 CPU with $2.8GHz$ and $4GB$ RAM. The laptop uses Ubuntu 12.10 Quantal Quetzal and also ROS Groovy Galapagos. Moreover, for obtaining sensor measurements a Kinect for Xbox 360 RGB-D camera is employed.

The implementations of KPIECE, the original STOMP and the new STOMP, which was implemented with regard to this thesis, are all modified for the motion planning tests. Both, KPIECE and the original STOMP are changed, so the planning is called consecutively for a given number of times, while keeping the entire algorithms and executions unaffected. Moreover, all three implementations output different generated data, such as the solution trajectories, the runtime of each planning cycle and call, respectively, the number of way-points, the duration of the trajectories, and whether the planning was successful or not. The output is saved in various log files and is processed subsequently. In order to have standardized experiments, the planners get the same start and goal configurations and the obstacles are positioned at the same location. Hence, a feature within the ROS planning pipeline is utilized to add an obstacle pole to the planning scene. Also, the configuration settings of each of the planners are adjusted, so the planning is not only executed for the same start and goal configurations, but also for the same kinematic chain.

6.3 Experiments

Overall, quantitative experiments are made and the quality of the underlying approach is tested. For the experiments the data is acquired as previously stated and the established criteria are applied for evaluation purposes.

Subsequently, I will describe the executed experiments in detail and also present the results of the experiments. Finally, I will discuss the findings and draw conclusions from them regarding the rating of the underlying approach of this thesis.

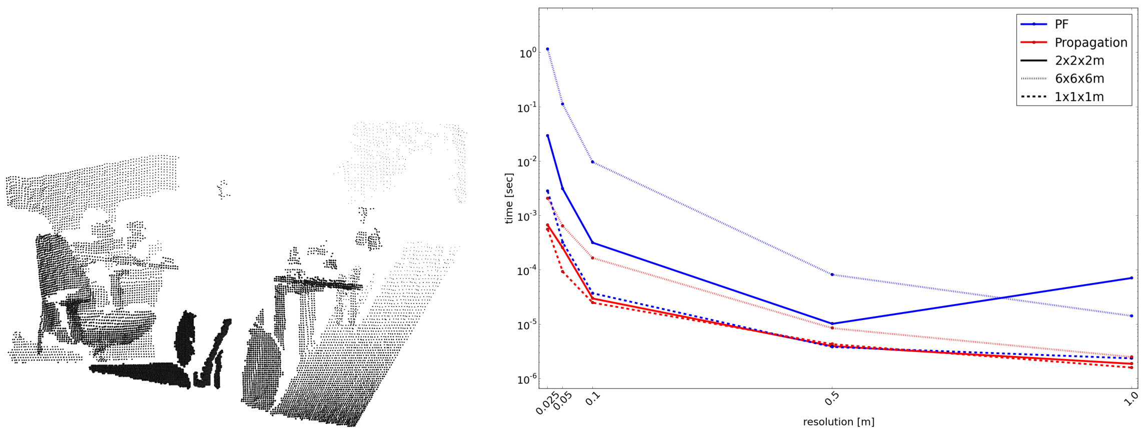
6.3.1 Distance Fields

Choosing the size and resolution of the environment representation depends mainly on the size of the workspace of the robot and the size of possible obstacles as well as the precision of the robot motions, respectively. Still, the decision is also up to the given possibilities for an environment representation and the performance.

As ROS provides two different classes for distance fields, namely the PF distance field and the Propagation distance field, it is reasonable to compare the performances of both representations in order to choose the superior class. The PF distance field employs a raster scanning type method in contrast to the Propagation distance field, which uses a vector propagation method. The runtime of both variations is measured while translating a given point cloud into a distance field. Thus, a point cloud of an office environment is obtained by means of an RGB-D camera (cf. Fig. 6.1a) and then transformed into a discretized voxel grid. The size and resolution of the voxel grid is similar to the values of both distance fields. Subsequently, each occupied voxel cell is transfered to the distance field and the times are measured for the whole process of adding the points to the distance field. The runtimes are determined for both classes for the resolutions $r = \{0.025, 0.05, 0.1, 0.5, 1.0\}m$ and for the grid sized of $1\,m \times 1\,m \times 1\,m$, $2\,m \times 2\,m \times 2\,m$, and $6\,m \times 6\,m \times 6\,m$.

The results for both types of distance fields is depicted in Fig. 6.1b. Clearly, the propagation distance field is overall faster than the PF distance field. Only with the smallest grid size, the PF distance field can keep up with the propagation distance field. Since the durations for the propagation distance fields with sizes $1\,m \times 1\,m \times 1\,m$ and $2\,m \times 2\,m \times 2\,m$ only slightly differ for the highest resolution, a propagation distance field with the scale of $2\,m$ and a little bit higher resolution of $0.015\,m$ is chosen for the implementation of the STOMP algorithm.

In addition to testing the performance of the two distance field versions, the runtime of creating and filling the distance field within the new STOMP implementation is measured.



(a) Test point cloud for creating distance field.

(b) Resulting times for initializing both distance field types.

Figure 6.1: (a) The test point cloud used for the evaluation of both distance field types. (b) The resulting runtimes for initializing both distance field types with different sizes and resolutions.

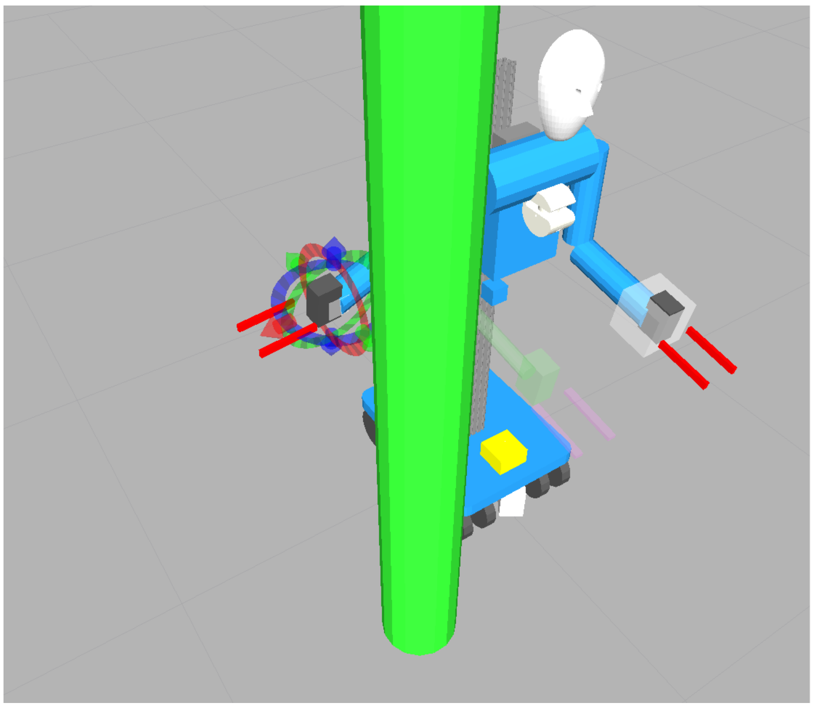
Since creating and filling the distance field within includes more actions with regard to the robot model and the robot state, the resulting runtimes differ from the determined durations from the previous experiments regarding the PF and Propagation distance fields.

The times are measured for 1000 planning cycles with and without an obstacle. The planning is done for the 7 joints of the arm, which means that several bodyparts have to be registered into the distance field in order to avoid self-collisions. The setting of the experiment is shown in Fig. 6.2.

Then, the average runtime of creating the distance fields is put in relation to the average overall runtime of the motion planning process and also compared to the average duration of the optimization process. As it is shown in Tab. 6.1, creating the distance field is, after the optimization process, the second most time-consuming process within the motion planning process. When distinguishing between tests with or without an obstacle, the impact of additional obstacles can be seen.

In case of no obstacle being present and only the bodyparts of the robot, which are not used for planning, having to be registered in the distance field, the runtime of creating and filling the distance field averages $0.07192\,s$ with a standard deviation of $\pm 0.00502\,s$. Compared to the runtime of the optimization the creation of the distance field only takes 0.6 times of the former. However, when an obstacle is added, the duration of creating the distance field increases to $0.10787\,s$ with a standard deviation of $0.00825\,s$, which can be translated to 0.93 times the runtime of the optimization process.

Overall, the time span for creating the distance field increases to a greater extent, while the runtime of the optimization process only gains about $0.00194\,s$ in average and thus, hardly changes when adding an obstacle.


(a) Setup for evaluation seen from the side.

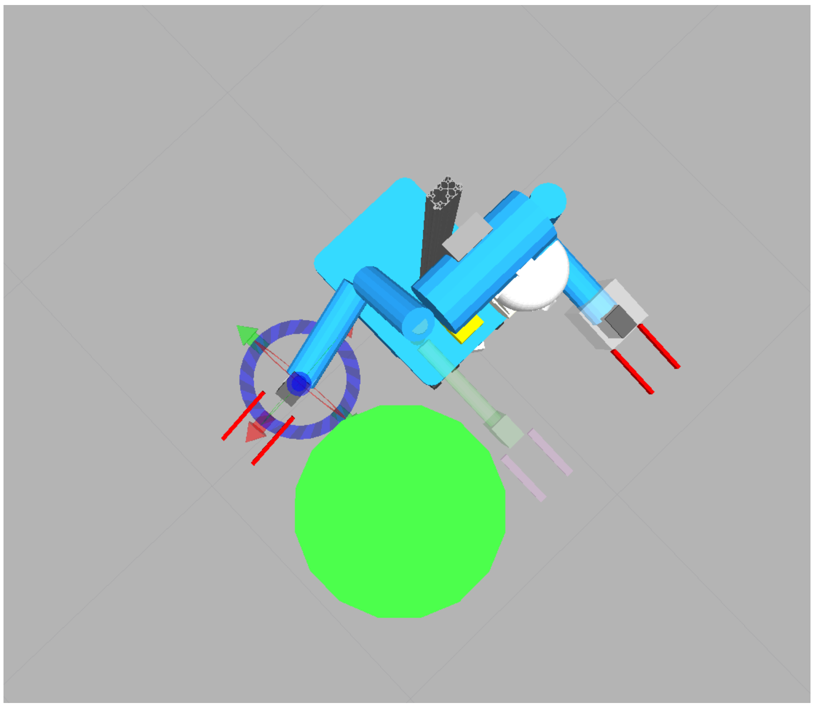

(b) Setup for evaluation seen from the top.

Figure 6.2: The start and goal configuration for the robot arm as well as an obstacle in terms of a cylinder for testing the three planning algorithms. The start configuration is shown by the see-through green and violet robot collision parts and the goal configuration is shown by the position of the arm and endeffector of the robot model.

Table 6.1: The parts of the optimization process and the distance field creation to the overall planning.

	multiresolutional STOMP	
	no obstacle	obstacle
Overall runtime $[s]$	0.18728	0.22521
Optimization runtime	61.12%	51.69%
Distance field creation	38.40%	47.90%
Remainder	0.48%	0.41%

6.3.2 Comparison of KPIECE, STOMP and the new Approach

The most interesting question with reference to the new approach for improving the STOMP motion planner presented in this thesis, is: How well does the improved planner do with regard to runtime and planning successful plans? In order to have a basis for rating the new planner, it has to be compared to known and already existing motion planners. As motion planning with KPIECE is already implemented within the motion planning module for Cosero and Dynamaid, it is reasonable to contrast it to the new STOMP planner. Also, the original STOMP implementation is available and allows to directly analyze whether the modifications to the original implementation really improved the motion planning with the STOMP algorithm. Accordingly, in most of the experiments the new STOMP is checked against the given KPIECE implementation and the original STOMP implementation.

In order to allow for structured experiments, a planning setup was defined by means of the planning components visualizer. Start and goal configurations as well as the position of an obstacle pole were determined. Eventually, the planning problem depicted in Fig. 6.2 was pre-defined and used for testing all of the three planners and comparing the results. Two different cases within this planning setup were used. First the planning was executed without the obstacle and afterwards, the planning was executed again with the pole positioned as it is seen in Fig. 6.2. The start and goal configurations were kept throughout both experiments. In this section the planning is conducted only for the 7 joints of the robot arm and executed for 1000 planning cycles.

During the execution of all of the three planners several values were outputted and saved. In order to compare the planners to each other, the data was post-processed and evaluated. The categories for evaluation include the success rate of the planners, the average number of waypoints, the duration of the trajectories, the average overall runtime of the motion planning processes as well as its standard deviation, and the time span for post-processing. With regard to the original and the new STOMP implementations, the average number of iterations until convergence was computed, too. The results of the experiments are listed in Tab. 6.2.

Both STOMP implementations were able to always find feasible and collision-free trajectories regardless of an obstacle being present. In contrast, KPIECE was indeed able to always find paths with no obstacle in the vicinity of the robot, but the success rate dropped to 823 feasible and collision-free paths out of 1000 when an obstacle was added. In principle, both STOMP implementations as well as the KPIECE implementation employ terminat-

Table 6.2: Comparison of KPIECE, original STOMP and the improved STOMP over 1000 runs for the 7 joints of the arm.
(*) Pre-defined runtime of the trajectory filter.

	KPIECE		original STOMP		new STOMP	
	no obstacle	obstacle	no obstacle	obstacle	no obstacle	obstacle
Success	100%	82.3%	100%	100%	100%	100%
No. of waypoints	54	125.626	101	101	27	27
Duration	n/a	n/a	$5\,s$	$5\,s$	$4.55\,s$	$4.55\,s$
Iterations	n/a	n/a	19	20.487	19	19.007
Overall runtime [s]	0.03885 $\pm$0.01248	1.42611 $\pm$1.22194	0.32843 $\pm$0.01613	0.38085 $\pm$0.02157	0.18728 $\pm$0.00205	0.22521 $\pm$0.02208
Post-processing	$1\,s$*	$1\,s$*	$0\,s$	$0\,s$	$0\,s$	$0\,s$

ing conditions and subsequently, report the failure when those terminating conditions are met. In case of STOMP the optimization process is aborted when the maximum number of iterations, concretely being 500 iterations, is reached. Instead, KPIECE uses a time limit, which is set to $5\,s$ in the current configuration, and aborts the planning process when the time limit is exceeded. Thus, KPIECE was able to only find solutions within the time limit for 82.3% of the planning cycles.

The number of waypoints of the found trajectories is different due to the characteristics and implementations of the planners. While the number of waypoints in the original STOMP implementation is fixed and set to 100 waypoints not including the goal, the new STOMP implementation calculates the number of waypoints for each new planning problem. Similarly, the duration of the trajectory is pre-defined to $5\,s$ in the original STOMP and calculated as shown in Eqn. 5.1. Accordingly, the duration of the trajectory in this experiment is $4.55\,s$ and by means of the defined growth function for the multiresolutional waypoint-spacing the number of waypoints equals 27. Unlike STOMP, KPIECE does not provide the duration of the trajectory. The number of waypoints is determined by chance, as the KPIECE algorithm only searches for a feasible and collision-free path in the configuration space and picks the first one, which is found. Therefore, it is a coincidence associated with the exploration of the search. Since finding a path from the start to the goal configuration without any obstacles is a simple problem, KPIECE directly finds a solution with 54 waypoints. When an obstacle is present, the average number of waypoints changes to 125.626.

Comparing the average number of iterations needed for the optimization to converge, both STOMP implementations only need 19 iterations when no obstacle is placed in the vicinity of the robot. When the obstacle cylinder is added, the improved STOMP implementation only needs 26 iterations in the first planning cycle and then settles with 19 iterations for the rest of the planning. In contrast, the original STOMP needs an average of 20.487 iterations to converge. This is presumably due to the initialization of the new STOMP implementation, which might be confirmed in Section 6.3.5.

When looking at the average overall runtime of all of the three motion planners, it can be seen that KPIECE is the fastest planner without obstacles, but becomes the slowest planner with obstacles in presence. Furthermore, unlike both STOMP implementations, KPIECE depends on a post-processing step executed by a trajectory filter module in order to reject redundant waypoints, smooth the trajectory and also determine the duration of the planned trajectory. The duration of the post-processing step can be pre-defined with regard to the desired precision. Within the current motion planning for the robots Cosero and Dynamaid the duration of the trajectory filtering is set to 1 s. Although the runtime of the post-processing can be chosen arbitrarily, the post-processing is necessary to get a smooth trajectory and therefore, the time span for executing the trajectory filter has to be added to the average overall runtime of the KPIECE motion planner. As a result, KPIECE is clearly slower than both STOMP motion planners in both cases, with or without an obstacle in the way.

In comparison to the original STOMP implementation, the new approach only needs 0.57 times and 0.59 times of the average overall runtime of the original implementation without and with an obstacle, respectively. The lower runtime in case of an obstacle being added can be associated with the average number of iterations, which is higher for the original STOMP.

Overall, the new approach for continuous replanning with STOMP and with multiresolution in time is an improvement of the STOMP implementation with regard to the established criteria in this experiment.

An example for trajectories for each of the 7 joints in the robot arm planned by KPIECE and both STOMP implementations is depicted in Fig. 6.3, Fig. 6.4 and Fig. 6.5. The blue trajectories are planned by KPIECE, whereas the green trajectories are determined by the original STOMP and the magenta trajectories are provided by the improved STOMP with a multiresolutional waypoint-spacing. Moreover, the red trajectories are determined by the new STOMP utilizing a uniform growth function and therefore, employing equidistantly spaced waypoints. Since KPIECE does not provide the duration of the trajectory and therefore, no discretization, too, the discretization in the graphs is chosen with reference to the resolution of the other trajectories. This is adequate enough for an example of the trajectories calculated by KPIECE. Apart from this, the trajectories determined by KPIECE are very noticeable, as they are not smooth due to no post-processing. This stresses the fact that post-processing by means of a trajectory filter is inevitable.

Altogether, the four trajectories slightly resemble each other for many joints. However, the trajectories determined by KPIECE sometimes follow different paths in joint space. The three STOMP trajectories choose a similar path in joint space except for the wrist pitch joint (cf. Fig. 6.5b), where the original STOMP implementation chooses a contrasting path resembling the KPIECE path.

6.3.3 Finite Differencing

In order to calculate the control cost matrix for the multiresolutional STOMP, it is necessary to employ the finite differencing algorithm by Fornberg as described in Section 5.5.4. The whole process of determining the finite differencing matrix consists of creating the α_ν vector out of the previously calculated multiresolutional time intervals and then executing the algorithm showed in Alg. 2. In addition, the filters resulting from the algorithm have to

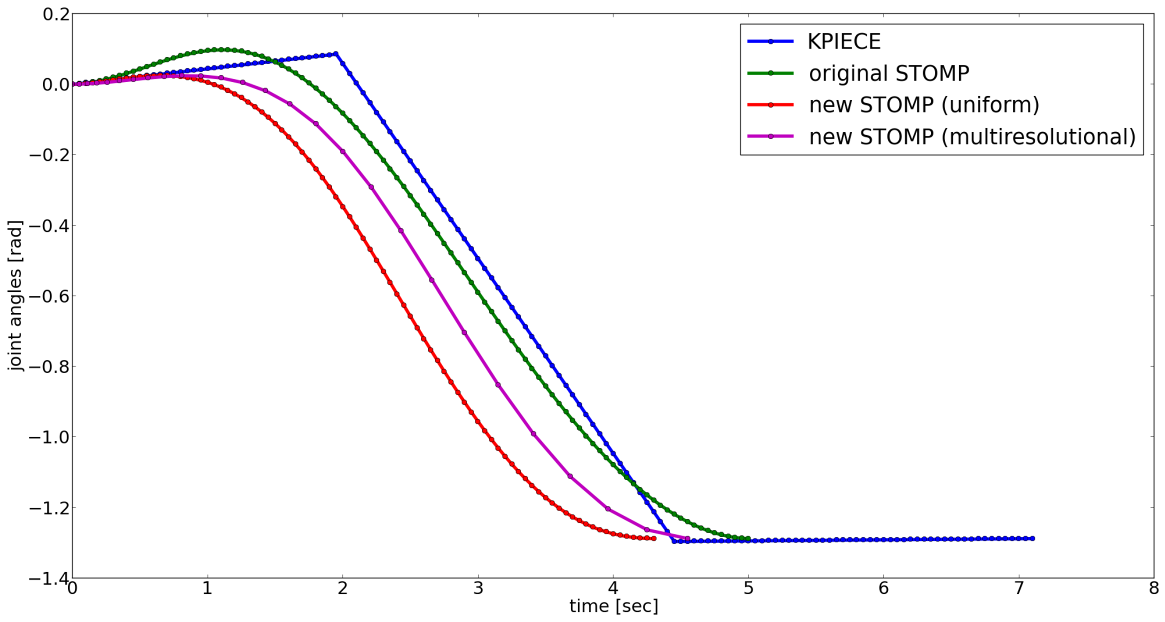


Figure 6.3: The resulting trajectories for the shoulder yaw joint while avoiding an obstacle. The trajectories are determined by KPIECE (blue), the original STOMP (green), and the new STOMP implementation with uniform waypoint spacing (red) and multiresolutional waypoint spacing (magenta).

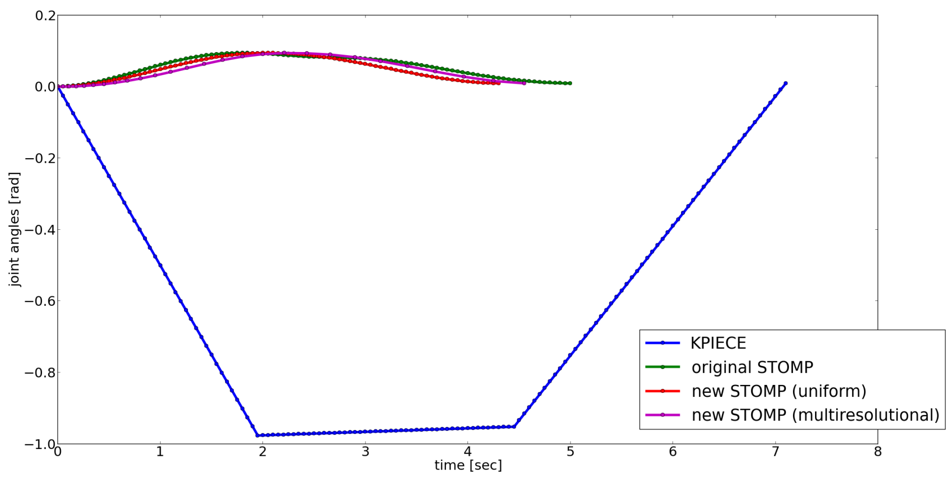
be sorted for further processing, as they are saved in the order of α_ν, that is the point for which the derivatives are calculated being at the beginning. Accordingly, the filters are sorted chronologically in terms of the given trajectory and its resolutions.

In this experiment this whole process is executed for multiresolutional trajectories with the growth function from Eqn.5.2. The size of the trajectories range from 10 to 100 waypoints and altogether, the algorithm is executed for each trajectory for 25 times in order to measure the performance of the algorithm.

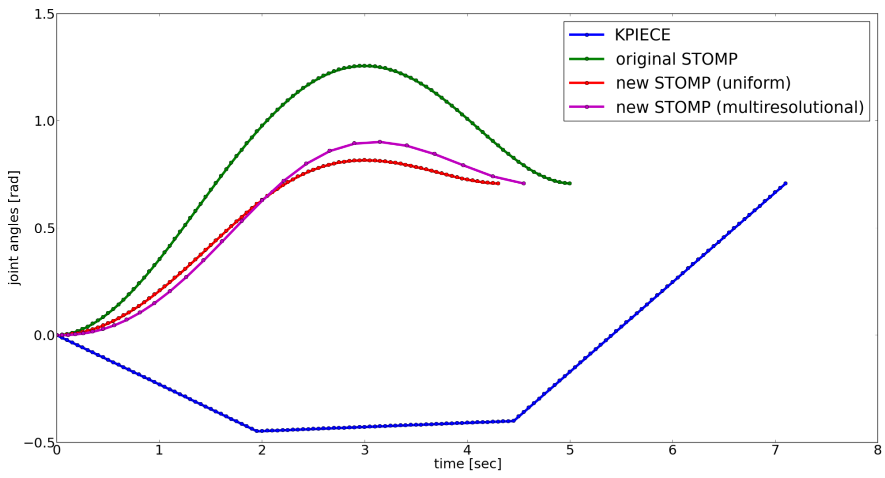
The results of the measurement are depicted in Fig. 6.6 in terms of the mean runtime for each number of waypoints as well as the covariance in Fig. 6.6a and the standard deviation in Fig. 6.6b. Both are depicted because, due to the nature of the results, it is difficult to show the gain of the runtime and the variation simultaneously.

As seen in the graphs the overall runtime of the finite differencing algorithm is constantly low and only increases lightly with a rising number of waypoints. There are a few peaks with increased runtime, but since those peaks only happened during one testrun each, those measurements can be labeld as outliers. This means that the rapid rise of runtime was affected by external influences such as the process managing within the desktop pc used for experiments.

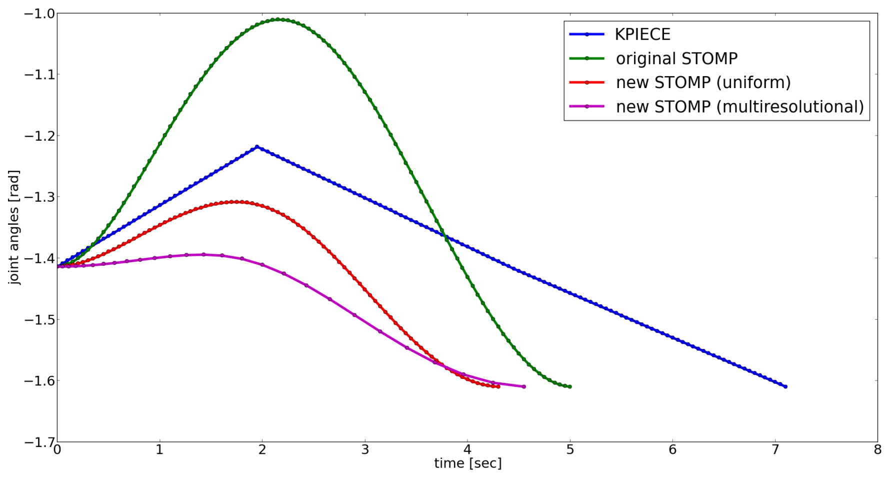
Overall, having to employ this algorithm for enabling muliresolutional motion planning is not costly with reference to the overall runtime of the STOMP motion planning. This is reassured by the results seen in Tab. 6.1, since the finite differencing algorithm is included in "Remainder".



(a) Trajectories for shoulder roll.

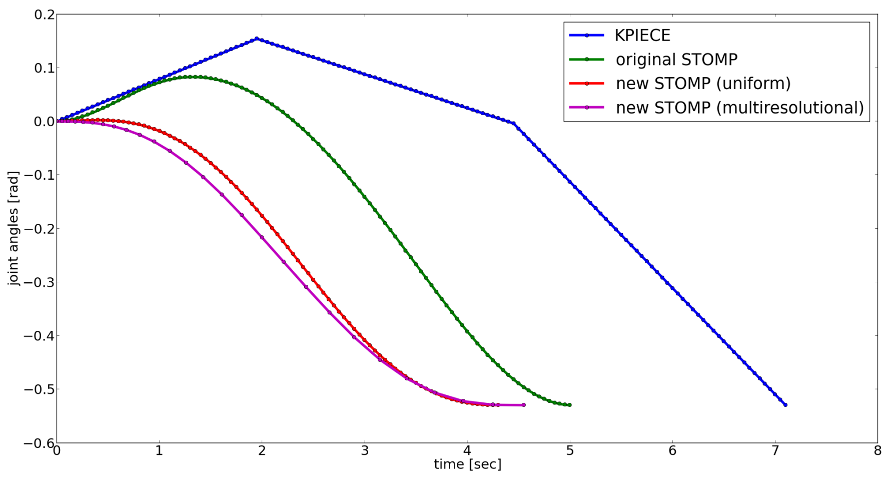


(b) Trajectories for shoulder pitch.

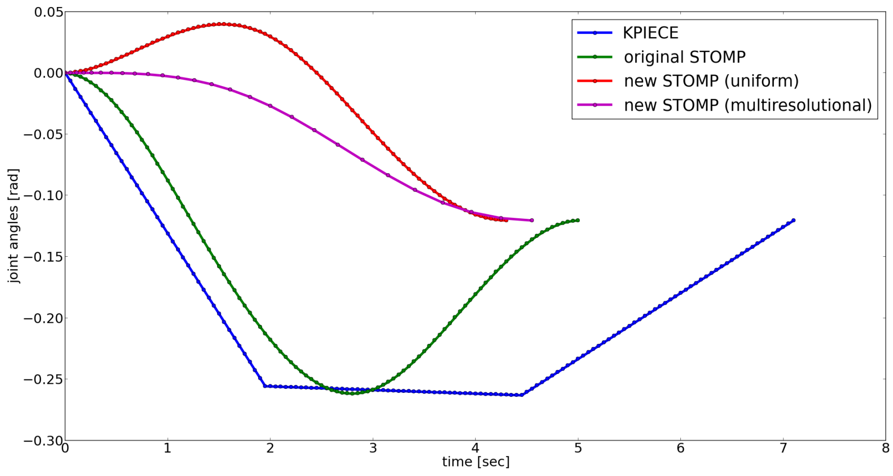


(c) Trajectories for elbow pitch.

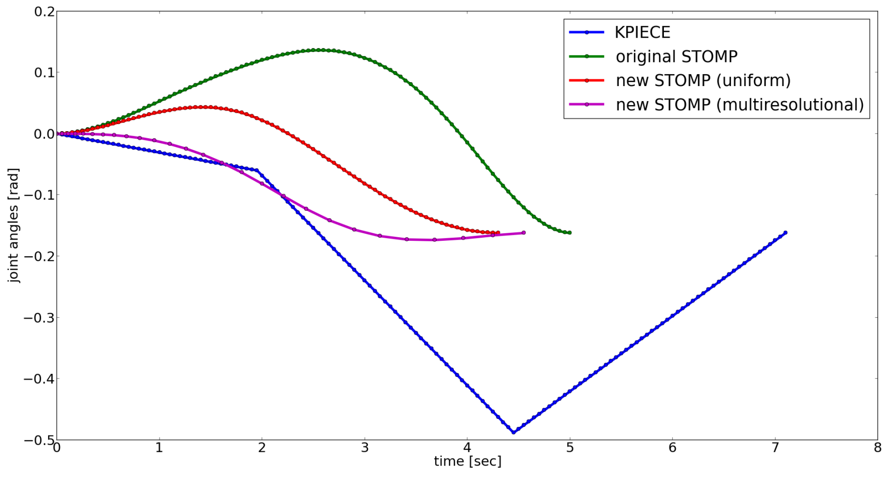
Figure 6.4: The resulting trajectories for the shoulder pitch and roll joints as well as the elbow pitch joint while avoiding an obstacle. The trajectories are determined by KPIECE (blue), the original STOMP (green), and the new STOMP implementation with uniform waypoint spacing (red) and multiresolutional waypoint spacing (magenta).



(a) Trajectories for wrist yaw.

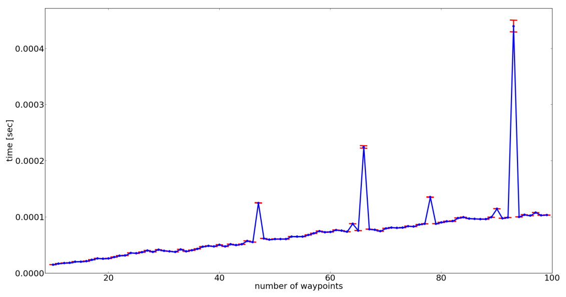


(b) Trajectories for wrist pitch.

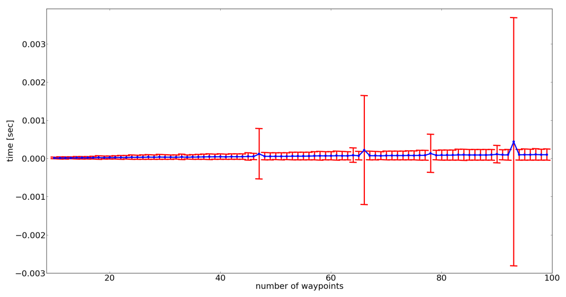


(c) Trajectories for wrist roll.

Figure 6.5: The resulting trajectories for the wrist yaw, pitch and roll joints while avoiding an obstacle. The trajectories are determined by KPIECE (blue), the original STOMP (green), and the new STOMP implementation with uniform waypoint spacing (red) and multiresolutional waypoint spacing (magenta).



(a) Mean and covariance.



(b) Mean and standard deviation.

Figure 6.6: The mean (blue) of the runtimes of the finite differencing algorithm by Fornberg for 10 to 100 waypoints together with (a) the covariance (red) and (b) the standard deviation (red).

6.3.4 Multiresolution versus Uniform Waypoint-Spacing

Not only is it necessary to compare the new STOMP implementation to the original one, but also with regard to the multiresolutional approach it is very interesting to see, whether the multiresolutional approach decreased the overall runtime of the STOMP motion planner. Therefore, the multiresolutional waypoint-spacing is directly compared to the equidistant spacing.

Since the new STOMP implementation utilizes a growth function, which is set to Eqn. 5.2 for the multiresolutional time intervals, it can be changed to a uniform growth function $f(x) = rx$ with r being the resolution. Hence, the effect of the multiresolution can easily be determined.

The experiment is yet again done for the planning setup shown in Fig. 6.2. The arm motion planning is executed for 1000 planning cycles with and without an obstacle. As shown in Tab. 6.3 both, the uniform and multiresolutional approach, yield to a success rate of 100%. Due to the number of waypoints being calculated regarding the duration of the trajectory and the discretization, the number of waypoints for both approaches differ. The uniform trajectory consists of 87 timesteps, whereas the multiresolutional equivalent only has 27 timesteps. The duration of the trajectory is determined by the sum of Euclidean distances between the start and the goal configuration of each joint. As the duration is can be divided by the uniform discretization without a remainder, the duration of the uniform trajectory does not need to be altered. In contrast, the multiresolutional trajectory duration has to be changed to fit in the multiresolutional time intervals. Thus, the durations differ by $0.25\,s$.

Regarding the average number of iterations needed for the optimization to converge, both versions of the planner match. Not only do they have the same average number of iterations, but also in case of an obstacle being present both panners need 26 iterations in the first planning cycle and then manage converging on 19 iterations.

When examining the overall runtimes, it can be observed, that they scale with the number of waypoints. Overall, although both planners have the benefits of the new STOMP implementation, the multiresolutional approach only takes 0.45 and 0.50 times of the runtime of the uniform approach, respectively.

One example of a full trajectory provided by the new STOMP implementation with a

Table 6.3: Comparison of the uniform and multiresolutional improved STOMP over 1000 runs for the 7 joints in the arm.

	uniform STOMP		multiresolutional STOMP	
	no obstacle	obstacle	no obstacle	obstacle
Success	100%	100%	100%	100%
No. of waypoints	87	87	27	27
Duration	4.3 s	4.3 s	4.55 s	4.55 s
Iterations	19	19.007	19	19.007
Overall runtime [s]	0.41483 ±0.04604	0.45322 ±0.04755	0.18728 ±0.02047	0.22521 ±0.02208

uniform growth function as well as a multiresolutional is shown in Fig. 6.3, Fig. 6.4 and Fig. 6.5. In the graphs the uniform trajectory is colored red and the multiresolutional trajectory is depicted in magenta. In this example it can be observed, that the multiresolutional trajectory has a longer duration as also shown in Tab. 6.3 and, while overall both trajectories have a light resemblance, the resulting trajectories differ in terms of changes in joint angles.

6.3.5 Initialization

In order to evaluate the approach of using a previously planned trajectory as an initialization of the optimization process, the new STOMP implementation is compared to a version of it with the standard initialization used by the original STOMP. That is an interpolation between the start and the goal configuration by means of Bézier splines. Since utilizing recent trajectories as an input for optimization is most reasonable in combination with frequent replanning, the experiments focus on dynamic obstacles. Altogether, two different kinds of experiments are executed. The established criteria for these experiments are the rate of success, the average number of iterations as well as the average overall runtime of the planning.

The first experiment is depicted in Fig. 6.7. The experiment starts with planning without an obstacle in the vicinity of the robot. In the next step, an obstacle is manually placed directly in the way of the planned trajectory and after a short time of adjusting to the obstacle, it is removed again. This is repeated for about 5 times within a total of 500 planning cycles. Since the dynamic obstacle is placed manually, the planning cycles with or without an obstacle being present differ between both versions of the planner. Still, the behavior is comparable.

The results of the first experiment are shown in Tab. 6.4. While the success rate of the STOMP version with initialization is 100%, the planner without using the recent trajectory as an initialization only has a success rate of 97.8%. Also, the average number of iterations needed until convergence is higher for the planner without initialization. As a result, the average overall runtime of the STOMP implementation with initialization is lower than for

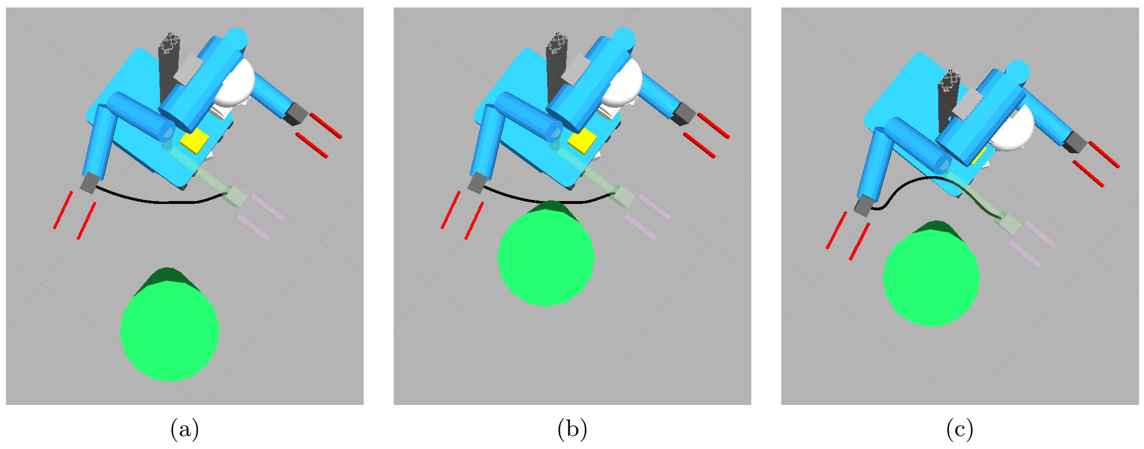


Figure 6.7: The avoidance of a dynamic obstacle by means of frequent replanning. The obstacle is directly placed in the vicinity of the trajectory in (b) and the corrected plan is shown in (c).

the other version of the planner.

The number of iterations for both versions of the planner throughout the 500 planning cycles are depicted in Fig. 6.9a. The STOMP version, which utilizes recent trajectories as an input for optimization is depicted in red and the version without initialization is shown in blue.

When looking at the graph it is noticeable that in the last two phases, in which the obstacle was standing in the vicinity of the robot, sometimes the number of iterations equals -1. This indicates that no feasible and collision-free trajectory was found in this planning cycle. Since the failures occur spreaded, it means that the planner, which does not utilize previously planned trajectories as an initialization, struggles to optimize the standard initial trajectory to a collision-free trajectory. Moreover, the number of iterations of the planner without initialization is constantly high when the obstacle blocks the way. In contrast, the version of STOMP, which utilizes recent trajectories as an input, only has one peak in the number of iterations at the beginning of each time the obstacle is placed in the vicinity of the robot. Otherwise the number of iterations for this version stays at 19 iterations.

The second experiment is depicted in Fig. 6.8. In the experiment, the dynamic obstacle is not directly placed in the way, but is slowly approaching the robot and subsequently leaves again. This is slowly done once within the 500 planning cycles. Again, the manually executed movements differ between both versions of the planner.

The results of the experiment are shown in Tab. 6.5. In case of the slowly approaching dynamic obstacle, both versions of the new STOMP motion planner find feasible and collision-free trajectories in each planning cycle. Nevertheless, both, the average number of iterations as well as the average overall runtime of the planner version without the initialization with recent trajectories, are higher than the respective values of the STOMP with initialization. The number of iterations for both versions of the planner throughout the 500 planning cycles are depicted in Fig. 6.9b. The STOMP version without utilizing the previously planned trajectory as an initial trajectory is shown in blue and the other version

Table 6.4: Comparison of the improved STOMP with and without using previously planned trajectories as an intialization. The obstacle is rapidly put in the way of the trajectory and then removed again after a while. This is repeated for about 5 times within a total amount of 500 runs.

	without initialization	with initialization
Success	97.8%	100%
No. of waypoints	27	27
Duration	4.55 s	4.55 s
Iterations	25.254	19.12
Overall runtime [s]	0.25116 $\pm$0.05271	0.21901 $\pm$0.01907

with initialization is depicted in red.

While the number of iterations of the STOMP version without initialization is high most of the time, in which the dynamic obstacle was moved to and away from the robot, the other planner again only shows only single peaks for every time a larger movement was executed. This indicates, that the planner using the standard initialization not only needs many iterations for adapting to an occurring obstacle, but also needs more iterations to return to the plan without an obstacle present.

Overall, the utilization of previously planned trajectories as an input for the optimization is effective with regard to lowering the number of iterations needed for the optimization to converge as well as decreasing the average overall runtime. Not only do the experiments presented in this section confirm this, but the impact of it can also be seen in the comparisons of the old STOMP implementation and the improved STOMP implementation with respect to the average number of iterations.

Furthermore, the experiments demonstrate, that the frequent replanning in order to avoid dynamic obstacles is successful especially when combined with the reuse of previously planned trajectories as an intialization to the optimization of the STOMP algorithm.

6.3.6 Additional joints

Up to now KPIECE, the original STOMP and the improved STOMP are only tested for the arm and the endeffector of the robot. Since it is possible to include other joints to the planning of all of the three motion planning implementations, it is reasonable to evaluate the performance of the motion planning algorithms with additional joints. As extra joints lead to a higher dimensionality of the c-space, the runtime of the planning is expected to increase with each additional joint. Therefore, the experiments are executed for the extra joints consecutively in order to observe the supposed gain of runtime.

As there are two torso joints, which can be added to the motion planning, first only the torso yaw joint is added and for a second experiment the torso yaw and the torso elevator are added. Similar to the setup of the motion planning only for the arm and endeffector, the start and the goal configurations are chosen by means of the planning components visualizer

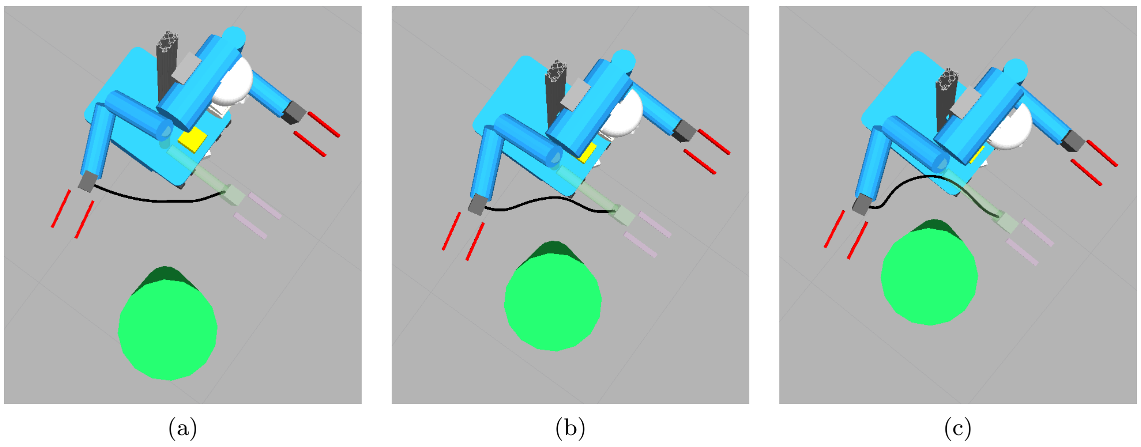


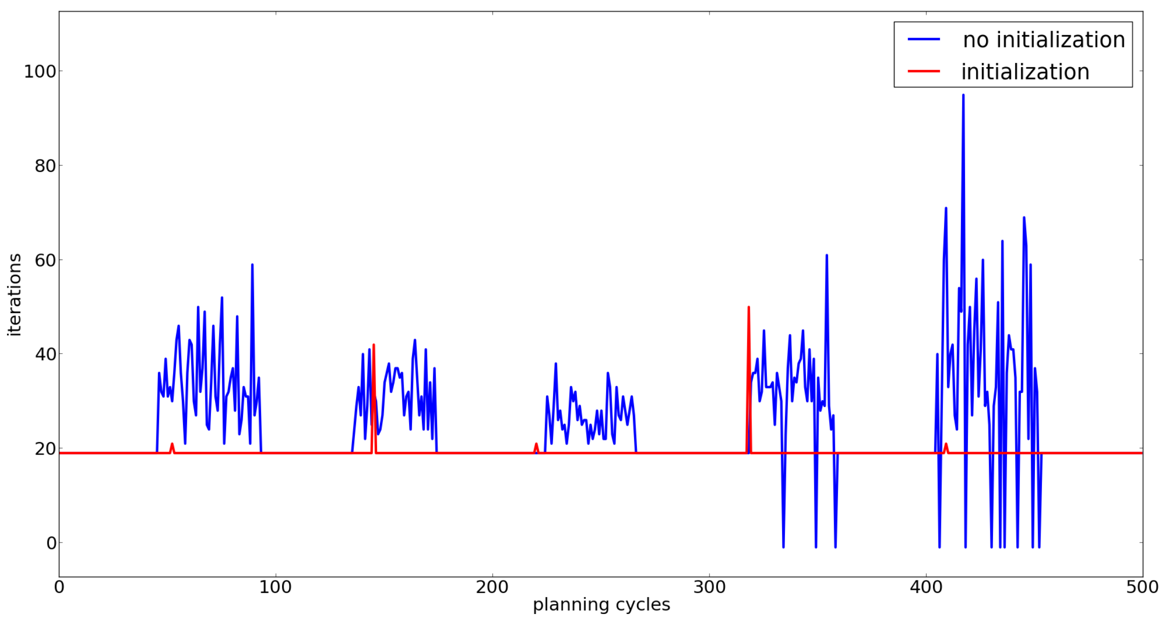
Figure 6.8: The avoidance of a dynamic obstacle by means of frequent replanning. The obstacle moves towards the robot and the plan is optimized with respect to the obstacle.

and also the positions of the obstacle are determined. The experiments are again executed for each of the three planners for 1000 planning cycles with and without the obstacle. Like the evaluation of the three motion planning implementations in Sec. 6.3.2 the motion planners are compared to each other in terms of the success rate, the average number of waypoints, the duration of the trajectories as well as the average overall runtime of the planners and the time span for post-processing. Again, both STOMP implementations are checked against each other regarding the average number of iterations needed for converging.

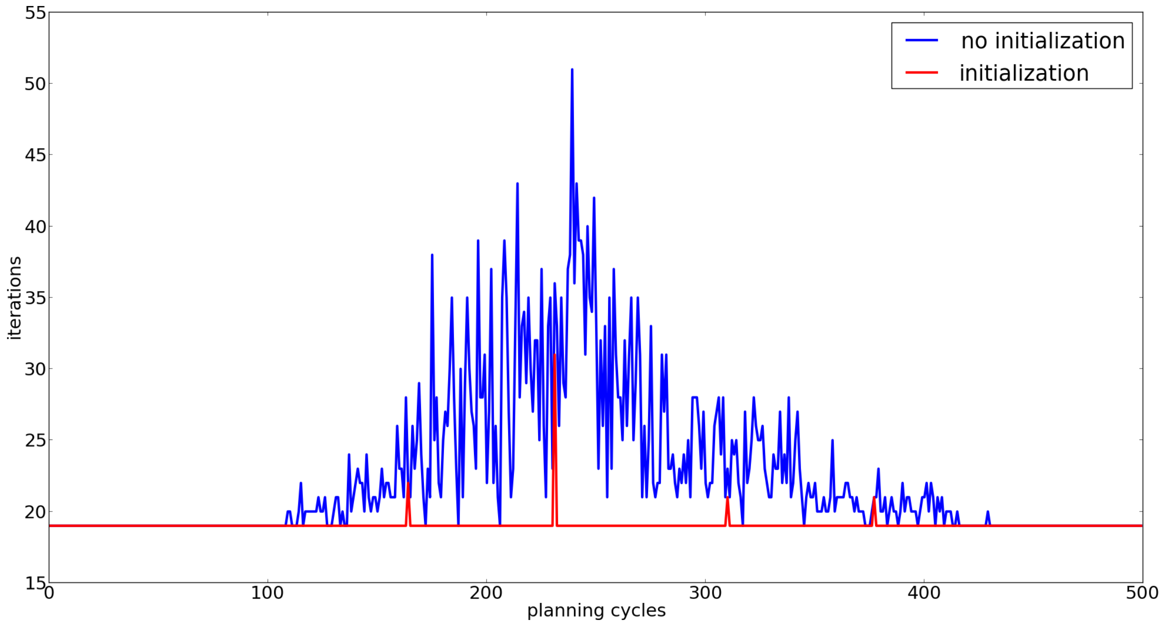
The setting for planning with the arm as well as the torso yaw joint is shown in Fig.6.10. In contrast to the setup for the experiments with the arm only, the goal configuration includes the torso facing towards the obstacle. The results of the experiment are shown in Tab. 6.6. While the STOMP implementations have a success rate of 100% for both, planning with and without an obstacle, KPIECE struggles to always find feasible and collision-free paths with an obstacle in the way. However, compared to only planning with the 7 joints of the arm, the success rate increases from 823 out of 1000 to a success rate of 92.2%. This might be due to the additional possibilities of finding a path by employing the extra joint.

Because of utilizing the additional joint, the Euclidean distance between the start and the goal configuration for each joint is lower compared to planning only for the arm, although the goal position of the endeffector is resembling in both cases. Therefore, the number of waypoints and the trajectory duration calculated by the new STOMP implementation decreases, while in the original STOMP implementation both values stay fixed. With regard to the average number of iterations, both STOMP implementations need 19 iterations for planning without an obstacle. While the new implementation of STOMP only needs 40 iterations for the first planning cycle and continues with 19 iterations with an obstacle in presence, the original STOMP implementation has an average of 29.174 iterations.

For the second experiment including the prismatic torso elevator joint, the setup was again changed. Here not only the torso faces the obstacle in the goal configuration of the robot, but also the elevator is used to lower the endeffector. The setting is depicted in Fig.6.11.

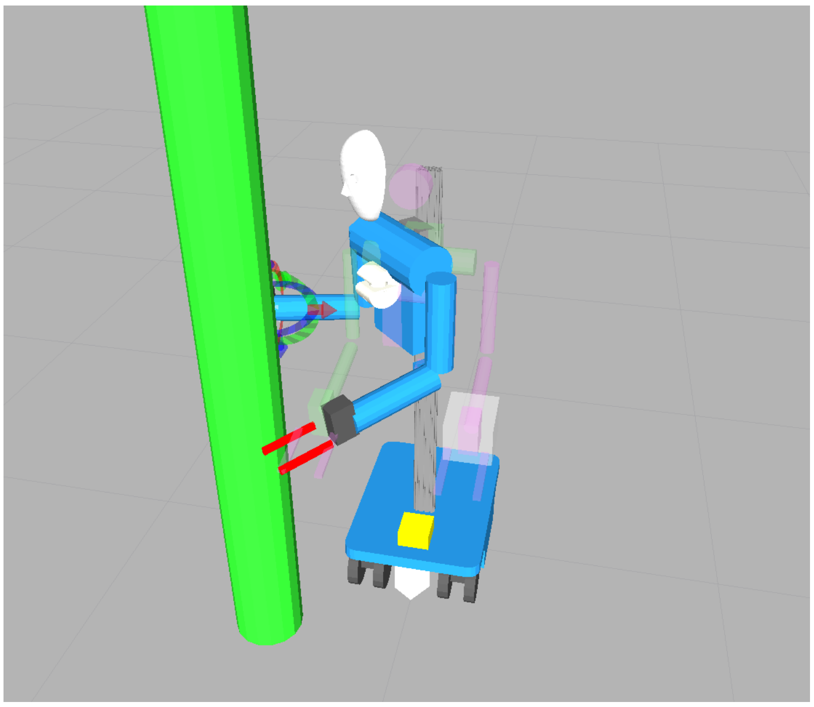


(a) Results for directly placing an obstacle in the vicinity of the trajectory and then removing it.

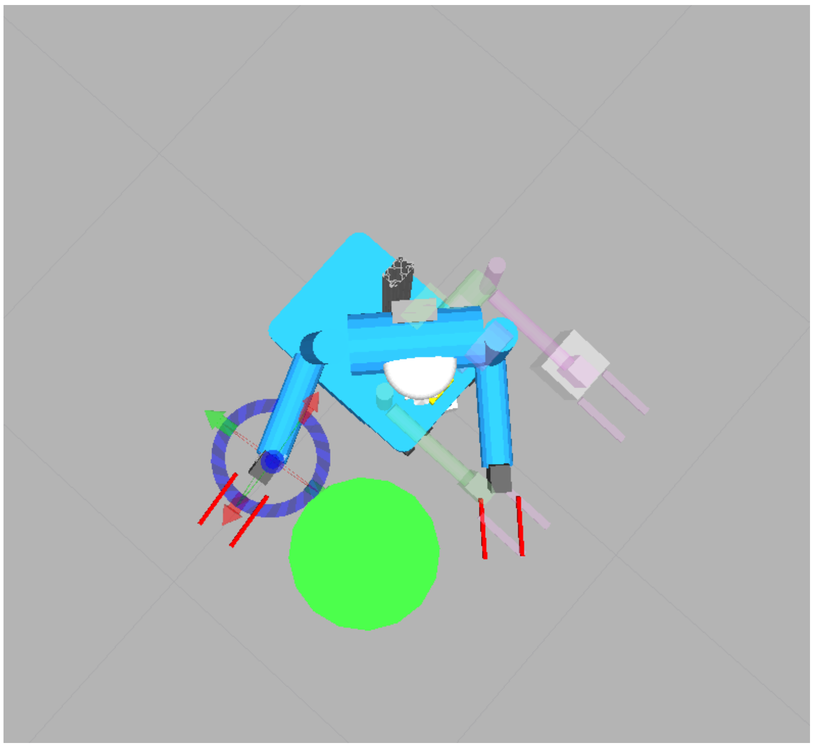


(b) Results for slowly moving an obstacle in and out of the vicinity of the trajectory.

Figure 6.9: The number of iterations until convergence for the improved STOMP without initializing with the recent trajectory (blue) and with initialization (red). In (a) the object was directly placed in the vicinity of the trajectory and in (b) the obstacle was slowly moved towards the robot. The experiments were executed for 500 planning cycles and the obstacle was placed 5 times and moved 4 times. The planning cycles with -1 iterations indicate, that no feasible and collision-free plan was found. The placing and moving of the obstacle was executed manually. Hence, the planning cycles with obstacles placed an removed differ slightly. The planning cycles in which the obstacles were placed is indicated by the peaks and increased number of iterations.

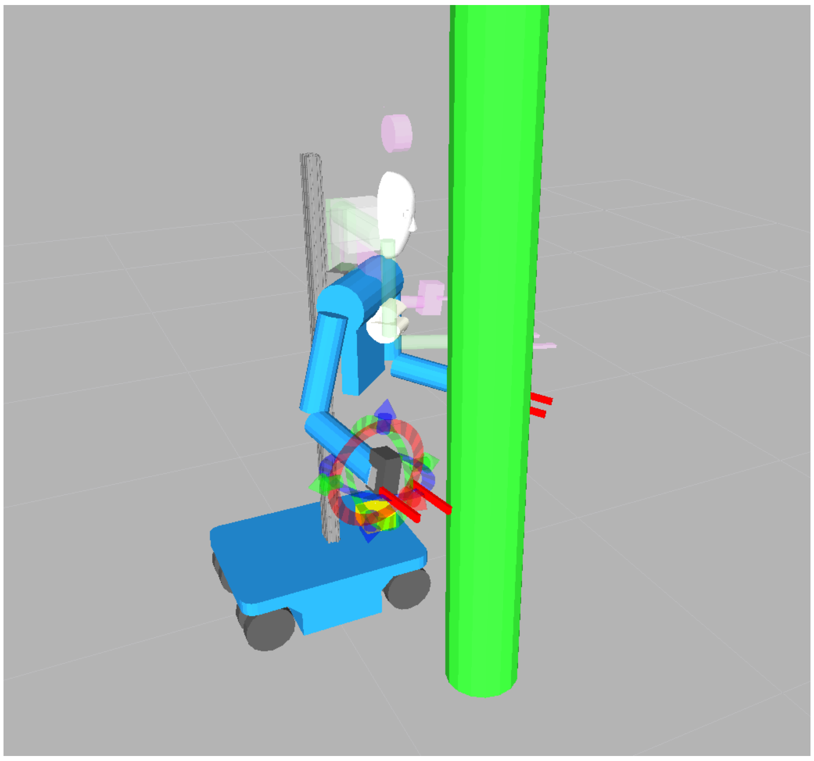


(a) Setup for evaluation seen from the side.

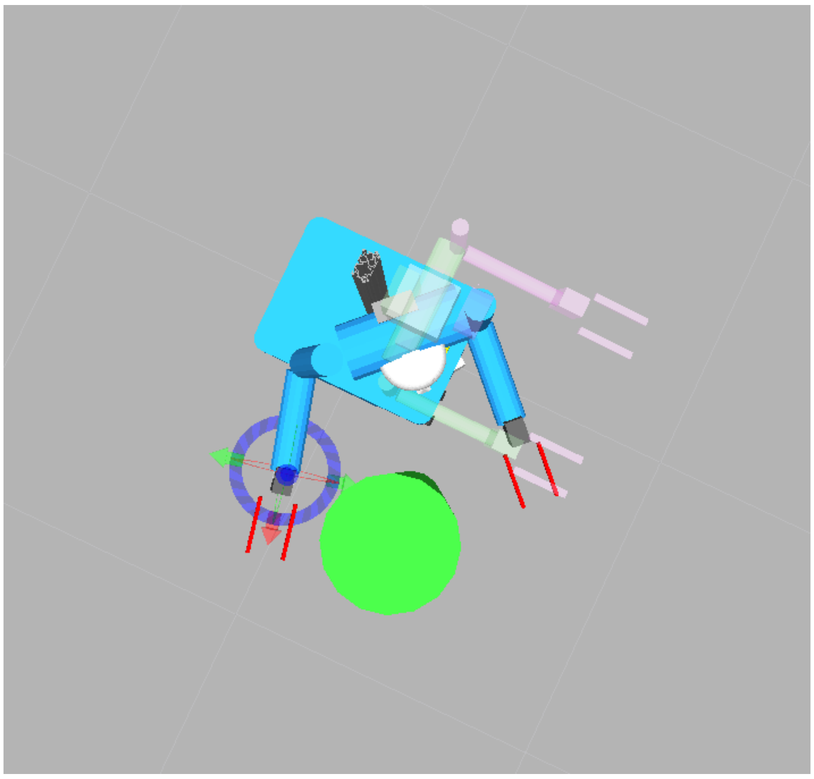


(b) Setup for evaluation seen from the top.

Figure 6.10: The start and goal configuration for the robot arm and torso joint as well as an obstacle in terms of a cylinder for testing the three planning algorithms. The start configuration is shown by the see-through green and violet robot collision parts.



(a) Setup for evaluation seen from the side.



(b) Setup for evaluation seen from the top.

Figure 6.11: The start and goal configuration for the robot arm, torso yaw joint and the elevator joint as well as an obstacle in terms of a cylinder for testing the three planning algorithms. The start configuration is shown by the see-through green and violet robot collision parts.

Table 6.5: Comparison of the improved STOMP with and without using previously planned trajectories as an intialization. The obstacle is slowly moved in the way of the trajectory and then slowly removed again after a while. This is repeated for about 4 times within a total amount of 500 runs.

	without initialization	with initialization
Success	100%	100%
No. of waypoints	27	27
Duration	4.55 s	4.55 s
Iterations	22.436	19.038
Overall runtime [s]	0.23464 ±0.03627	0.21604 ±0.02551

Since the arm joints and the torso yaw are revolute joints, it is interesting to utilize the three motion planners for planning with an additional prismatic joint. The results of the planning including the torso elevator are shown in Tab. 6.7. The STOMP implementations still find feasible and collision-free trajectories with and without an obstacle for each planning cycle. While the success rate of KPIECE for planning without an obstacle stays at 100%, the success rate drastically drops when an obstacle is added to the planning scene. In only 23.1% of the planning cycles, KPIECE was able to find feasible and collision-free paths within the given time limit of 5 s. Generally the planning in this case took relatively long with an average overall runtime of 1.96255 s without the additional 1 s for the trajectory filter. Although the planning problem got more difficult by adding the torso elevator, both STOMP implementations have overall performances comparable to both of their previous results, when planning only with the arm and with the additional torso yaw joint.

As shown in Tab. 6.8 the average overall runtime of KPIECE does not scale with regard to the number of joints and therefore the dimension of the search space. In contrast, the average runtime for planning with the original STOMP implementation increases with each additional joint in both cases of planning with or without an obstacle. Although it can be presumed that the improved STOMP implementation follows the original STOMP in scaling regarding the number of joints, it keeps its initially low average overall runtime throughout the experiments for the additional joints. It has to be noted, that the problems differ from each other, since the goal configurations of the additional joints do not base on the goal configuration of the arm motion planning. Still, the position of the endeffector is slightly resembling and it is possible to see the overall trend.

6.4 Discussion

Overall, the experiments support the approach for improving the STOMP motion planner presented in this thesis. The improved STOMP verifiably decreased the overall runtime for the motion planning process and thus, enabled employing frequent replanning and adding extra joints to the motion planning.

Table 6.6: Comparison of KPIECE, original STOMP and the improved STOMP over 1000 runs with the torso yaw as an additional joint.
(*) Pre-defined runtime of the trajectory filter.

	KPIECE		original STOMP		new STOMP	
	no obstacle	obstacle	no obstacle	obstacle	no obstacle	obstacle
Success	100%	92.2%	100%	100%	100%	100%
No. of waypoints	32	95.98	101	101	21	21
Duration	n/a	n/a	$5\,s$	$5\,s$	$2.9\,s$	$2.9\,s$
Iterations	n/a	n/a	19	29.174	19	19.021
Overall runtime $[s]$	0.06909 ±0.04916	1.03455 ±1.08000	0.48099 ±0.02322	0.79251 ±0.07025	0.19079 ±0.02231	0.22269 ±0.02150
Post-processing	$1\,s$*	$1\,s$*	$0\,s$	$0\,s$	$0\,s$	$0\,s$

The experiments with regard to the creation of the distance field showed, that the creation time increases drastically when obstacles are in the vicinity of the robot, while the average duration of the optimization process stays nearly the same. In order to further decrease the runtime of the overall planning, the focus should lie on a better approach for creating and filling the distance field and since the measured task includes additional functions within the implementation, those functions and also possible waiting times caused by locks should considered, too.

In contrast, the results of testing the runtime for the finite differencing algorithm show, that the calculation of the finite differencing matrix has very light effect on the overall runtime, which is dominated by the optimization process and the creation of the distance field. Therefore, employing multiresolution in time, for which this additional calculations are required, is not at all costly in runtime, but as shown in the other experiments yields to decreasing the overall runtime of the planning.

When comparing the new STOMP implementation with KPIECE and the original STOMP implementation, one of the results is, that the STOMP algorithm itself is superior to the KPIECE algorithm in terms of runtime and success rate. One of the benefits of STOMP is, that no further processing is needed, as the smoothness of the provided trajectory is already taken into account within the problem definition of STOMP. After further comparing the original STOMP implementation with the new STOMP implementing the presented approach in this thesis, it is illustrated that the changes lead to fewer iterations needed for the optimization to converge and therefore less overall runtime is needed for the planning with the improved STOMP motion planner. This is affirmed by the experiments regarding the reuse of previously planned trajectories as an initialization to the optimization. The experiments show that this approach yields to a somewhat faster runtime as well as a higher success rate concerning especially suddenly occurring dynamic obstacles. Furthermore, when checking the multiresolution in time against planning with a uniform waypoint-spacing, it is confirmed that the multiresolutional approach provides solution tra-

Table 6.7: Comparison of KPIECE, original STOMP and the improved STOMP over 1000 runs with the torso yaw and the torso elevator as additional joints.
(*) Pre-defined runtime of the trajectory filter.

	KPIECE		original STOMP		new STOMP	
	no obstacle	obstacle	no obstacle	obstacle	no obstacle	obstacle
Success	100%	23.1%	100%	100%	100%	100%
No. of waypoints	32	88.41	101	101	20	20
Duration	n/a	n/a	5 s	5 s	2.66 s	2.66 s
Iterations	n/a	n/a	19	26.597	19	19.019
Overall runtime [s]	0.03661 ±0.01454	1.96255 ±1.35102	0.51378 ±0.02551	0.82635 ±0.07755	0.19363 ±0.02482	0.22678 ±0.02400
Post-processing	1 s*	1 s*	0 s	0 s	0 s	0 s

Table 6.8: Comparison of KPIECE, original STOMP and the improved STOMP over 1000 runs. The average overall runtime is shown for planning only with the arm, with an additional torso yaw joint and finally also the torso elevator is added.

Overall runtime [s]	KPIECE		original STOMP		new STOMP	
	no obstacle	obstacle	no obstacle	obstacle	no obstacle	obstacle
Only arm	0.03885	1.42611	0.32843	0.38085	0.18728	0.22521
With torso yaw	0.06909	1.03455	0.48099	0.79251	0.19079	0.22269
With elevator	0.03661	1.96255	0.51378	0.82635	0.19363	0.22678

jectories with a comparable quality in lesser time.

Since the decreased runtime enables including additional joints to the planning process, it is tested whether the runtime scales with the increasing dimensions of the c-space. Although the planning problems differ, because the goal configurations regarding the experiments for the additional joints are not build on the goal configuration of the motion planning for the arm, a trend can be noticed. While the original STOMP scales with the increasing dimension, the improved STOMP implementation keeps a likewise performance. The experiments also show, that the overall runtime with additional joints is low enough to regularly include both joints to the motion planning. However, it is not reasonable to generally include the elevator joint, as it has special characteristics, which lead to the movements being time-consuming and costly. As it is the assignment of the robot control to decide whether it is reasonable to utilize the torso elevator and moreover, it is not possible to fully consider the characteristics of the torso elevator within the motion planning, the decision should be made outside the motion planning module. Nevertheless, the tests demonstrate, that the torso elevator can easily be included into the planning.

In conclusion to the experiments and evaluation, the changes referring to the presented approach have proven to be profitable. Moreover, the assumptions made concerning the ideas proposed in this thesis have been confirmed, since they effectively yield the intended behavior.

7 Summary and Future Work

In this thesis motion planning in the field of domestic service applications has been researched. An approach for continuous motion planning with multiresolution in time has been presented and evaluated. It is based on a motion planning algorithm called STOMP by Kalakrishnan et al. [8], which follows the approach of stochastic trajectory optimization. Thus, the motion planning algorithm uses an initial trajectory, which does not have to be collision-free, and iteratively minimizes the costs of the trajectory until convergence. In order to enable continuous planning to react to the dynamic household environment, the runtime of the STOMP motion planner had to be decreased. Hence, the concept of multiresolution in time was applied to the motion planner in terms of providing a multiresolutional trajectory representation and integrating it into the optimization process. In order to calculate the control costs for the multiresolutional trajectories a finite differencing algorithm was implemented and utilized to determine the coefficients for building the control cost matrix. Moreover, the frequent replanning was implemented and adjusted to the multiresolutional approach, so the planned trajectory becomes more refined after each planning cycle. Since the convergence of the STOMP algorithm depends on the initial trajectory, the initialization of the optimization process was changed in order to further decrease the runtime of the planner. Instead of using a newly created interpolation between the start and the goal configuration, the trajectory planned in the previous planning cycle is utilized as an input to the optimization process. Also in contrast to the original STOMP implementation, the duration of the trajectory is calculated for each planning problem by means of the body composition of the robot. Apart from this, additional joints of the robot's torso were added to the arm motion planning to lead to more solutions for given planning problems.

The presented approach was evaluated in experiments with regard to several established criteria like the overall runtime, the number of iterations, the calculated duration of the trajectory and the success in avoiding static and dynamic obstacles. It was compared to the original STOMP implementation, a uniform version of itself and moreover, to the current motion planning implementation with KPIECE. Overall, the experiments show that the improved STOMP implementation with continuous replanning and multiresolution in time is superior to the other planners with regard to the overall runtime and the number of iterations. Furthermore, the obstacle avoidance was successful in both cases and the benefits of the presented approach were demonstrated.

A possibility for future work is to extend the upper body motion planning presented in this thesis to a whole-body motion planning by combining it with the navigation of the basis. Since it is not reasonable to always plan for each body part in detail, the body parts of the robot could be grouped together to differently combined parts. When approaching a goal position the resolution of these groups could be gradually increased to lead to a smooth transition from coarse navigation for the whole robot to precise endeffector planning. Apart from this, the STOMP algorithm still has potential to decrease its runtime even further, when focusing on the integration of the environment representation in terms of the distance field. Moreover, a sequential planning for both arms could be integrated in terms of consecutively determining the goal configurations of both arms by means of inverse kinematics and subsequently solving the motion planning problem with STOMP. Finally, the improved STOMP implementation can be evaluated further in different scenarios and also on the real robot.

List of Figures

Bibliography

[1] Behnke, S. Local multiresolution path planning. In *In Proceedings of the 7th RoboCup International Symposium* (2003), Springer, pp. 332–343.

[2] Brock, O., and Khatib, O. Elastic strips: A framework for motion generation in human environments. *I. J. Robotic Res. 21*, 12 (2002), 1031–1052.

[3] Chitta, S., Jones, E. G., Ciocarlie, M. T., and Hsiao, K. Mobile manipulation in unstructured environments: Perception, planning, and execution. *IEEE Robot. Automat. Mag. 19*, 2 (2012), 58–71.

[4] Choset, H., Lynch, K., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L., and Thrun, S. *Principles of Robot Motion: Theory, Algorithms and Implementation.* MIT Press, 2005.

[5] Dietrich, A., Wimböck, T., Albu-Schäffer, A., and Hirzinger, G. Reactive whole-body control: Dynamic mobile manipulation using a large number of actuated degrees of freedom. *IEEE Robot. Automat. Mag. 19*, 2 (2012), 20–33.

[6] Fornberg, B. Generation of finite difference formulas on arbitrarily spaced grids. *j-MATH-COMPUT 51*, 184 (1988), 699–706.

[7] He, K., Martin, E., and Zucker, M. Multigrid chomp with local smoothing. In *Proceedings of 13th IEEE-RAS International Conference on Humanoid Robots (Humanoids)* (2013), IEEE.

[8] Kalakrishnan, M., Chitta, S., Theodorou, E., Pastor, P., and Schaal, S. Stomp: Stochastic trajectory optimization for motion planning. In *IEEE International Conference on Robotics and Automation, ICRA 2011, Shanghai, China, 9-13 May 2011* (2011), IEEE, pp. 4569–4574.

[9] LaValle, S. M. *Planning Algorithms.* Cambridge University Press, 2006.

[10] Nieuwenhuisen, M., Droeschel, D., Holz, D., Stückler, J., Berner, A., Li, J., Klein, R., and Behnke, S. Mobile bin picking with an anthropomorphic service robot. In *2013 IEEE International Conference on Robotics and Automation* (2013), IEEE, pp. 2327–2334.

[11] Park, C., Pan, J., and Manocha, D. Itomp: Incremental trajectory optimization for real-time replanning in dynamic environments. In *ICAPS* (2012), AAAI.

[12] Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software* (2009).

[13] Ratliff, N. D., Zucker, M., Bagnell, J. A., and Srinivasa, S. Chomp: Gradient optimization techniques for efficient motion planning. In *2009 IEEE International Conference on Robotics and Automation* (2009), IEEE, pp. 489–494.

[14] ROBOTIS. *ROBOTIS e-Manual.* `http://support.robotis.com/en/`.

[15] ROS. *Documentation KDL*, December 2013. `http://wiki.ros.org/kdl`.

[16] Stückler, J., and Behnke, S. Integrating indoor mobility, object manipulation, and intuitive interaction for domestic service tasks. In *Proceedings of 9th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, IEEE, pp. 506–513.

[17] Stückler, J., Gräve, K., Kläss, J., Muszynski, S., Schreiber, M., Tischler, O., Waldukat, R., and Behnke, S. Dynamaid: Towards a personal robot that helps with household chores. In *RSS 2009 Workshop on Mobile Manipulation in Human Environments* (2009).

[18] Şucan, I. A., and Kavraki, L. E. Kinodynamic motion planning by interior-exterior cell exploration. In *WAFR* (2008), vol. 57 of *Springer Tracts in Advanced Robotics*, Springer, pp. 449–464.

[19] Şucan, I. A., Moll, M., and Kavraki, L. E. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine 19*, 4 (December 2012), 72–82.

[20] Theodorou, E., Buchli, J., and Schaal, S. Reinforcement learning of motor skills in high dimensions: A path integral approach. In *ICRA* (2010), IEEE, pp. 2397–2403.

[21] Ye, Q.-Z. The signed euclidean distance transform and its applications. In *Proceedings of 9th International Conference on Pattern Recognition* (1988), pp. 495–499.