Institut für Informatik
der Technischen Universität München

Robot Motion Planning in Time-varying Environments

Andrea Baumann

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Chr. Zenger

Prüfer der Dissertation:

1. Univ.-Prof. Dr. H.-J. Siegert, em.
2. Univ.-Prof. Dr. Dr.h.c. W. Brauer

Die Dissertation wurde am 28. Juni 2001 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 19. Oktober 2001 angenommen.

Abstract

Motion planning is one of the principal tasks of autonomous robot systems. As a consequence, robot motion planning is one of the most active research areas in robotics and in the past decades great effort has been put into the development of flexible motion planning algorithms. The problems that needed to be tackled turned out to be demanding and most of the solutions found so far are restricted in the sense that they either find a collision free geometric path ignoring robot dynamics or they compute a time or energy optimal trajectory but they ignore collisions. The obvious goal, however, is to find a planning approach that respects both obstacles and robot dynamics. By including the time in the planning process it becomes possible to handle trajectories that are collision free and that also respect the dynamic limits of the robot. In addition, such a planning approach will be capable of dealing with time-varying obstacles. The position of such obstacles changes over time (but is known in advance for each point in time). Examples can be found in industrial manufacturing processes. Consider a robot in front of a production line with moving workpieces.

In this thesis we present such a planning approach. Our motion planning algorithm considers robot dynamics (i.e. force and torque limits) and is able to deal with time-varying obstacles. Another characteristic of our approach is that the planning process and the process of generating trajectories are decoupled. This bears the advantage that our planner can handle different types of trajectories without modification. For example, point-to-point motions, path motions, or even special types of trajectories for non-holonomic robots. The planning process itself only handles base points, which define allowed areas for the actual trajectory.

By modifying the position and the number of base points, the allowed trajectory area and consequently the trajectory itself is modified. To guide the planning process, we introduce several criteria upon which the evaluation of generated trajectories is based. Our main focus is on freedom from collision and robot dynamics, since these criteria have precedence over all other criteria, such as time or energy consumption.

One of the main ingredients of motion planning in time-varying environments is a reliable algorithm for collision detection. We present an extension of an existing algorithm

for static environments that enables us not only to detect collisions with moving obstacles but also gives us a precise rating of the depth of a collision.

To demonstrate the usefulness of our approach, we have implemented our motion planning algorithm within the scope of a robot simulation system and we have tested it in various scenarios.

Kurzfassung

Die Planung von Bewegungen ist eine der wichtigsten Aufgaben eines autonomen Robotersystems. In der Robotik-Forschung ist daher die Bewegungsplanung eines der Hauptthemen, und es wurden in den vergangenen Jahrzehnten große Anstrengungen unternommen, flexible Planungsverfahren zu entwickeln. Die dabei zu lösenden Probleme erwiesen sich als anspruchsvoll, und die meisten der existierenden Ansätze beschränken sich entweder auf das Finden von kollisionsfreien geometrischen Bahnen ohne Berücksichtigung der Roboterdynamik, oder die Berechnung einer Zeit- oder Energie optimalen Trajektorie ohne Berücksichtigung von Hindernissen. Der Vorteil eines Bewegungsplaners, der sowohl die Hindernisse im Raum als auch die Einhaltung der dynamischen Grenzen des Roboters beachtet, liegt auf der Hand. Durch den Einbezug der Zeit wird es möglich, Trajektorien zu berechnen, die sowohl kollisionsfrei sind, als auch die Dynamik des Roboters berücksichtigen. Hinzu kommt, dass es mit einem solchen Bewegungsplaner möglich wird, zeitvariante Hindernisse in den Planungsvorgang miteinzubeziehen. Diese Hindernisse haben die Eigenschaft, dass ihr Aufenthaltsort sich mit der Zeit ändert (zu jedem Zeitpunkt aber bekannt ist). Dies trifft zum Beispiel auf Szenarien zu, wie sie aus der Automobilindustrie bekannt sind. Man denke dabei an einen Roboter am Fließband, auf dem sich Werkstücke bewegen.

In der vorliegenden Arbeit wird ein solches Verfahren zur Planung von Roboterbewegungen vorgestellt. Der zugrundeliegende Bewegungsplaner bezieht Roboterdynamik und zeitvariante Hindernisse in den Planungsprozess mit ein. Eine weitere Besonderheit des vorgestellten Verfahrens ist, dass die Trajektorienerzeugung vom eigentlichen Planungsvorgang entkoppelt ist. Dies hat den Vorteil, dass der Planungsvorgang über verschiedene Trajektorientypen ablaufen kann. Seien das nun Punkt-zu-Punkt Bewegungen oder überschliffene Bahnen, oder auch spezielle Trajektorien für nichtholonome Roboter. Die Planung selbst findet nur über Stützpunkte statt, die erlaubte Bereiche für die Trajektorie definieren.

Durch das Modifizieren der Position und der Anzahl der Stützpunkte wird der erlaubte Trajektorienbereich und damit auch die Trajektorie selbst verändert. Um den Fortgang der Planung zu beurteilen, werden verschiedene Kriterien eingeführt, nach denen die gener-

ierte Trajektorie bewertet wird. Besonderen Augenmerk legen wir hierbei auf Kollisionsfreiheit und die dynamischen Randbedingungen, da eine Einhaltung dieser Kriterien als vorrangig anzusehen ist.

Bei einer Bewegungsplanung mit zeitvarianten Hindernissen ist ein wichtiger Bestandteil die Erkennung von Kollisionen des Roboters. Wir erweitern dazu ein bestehendes Verfahren um die Möglichkeit der Kollisionserkennung in zeitvarianten Umgebungen, und stellen einen Algorithmus vor, der neben der eigentlichen Erkennung von Kollisionen auch noch unmittelbar eine Bewertung der Kollisionstiefe erlaubt.

Um die Praxistauglichkeit unseres Verfahrens unter Beweis zu stellen, wurde der Planungsalgorithmus im Rahmen eines Robotersimulationssystems implementiert und anhand verschiedener Szenarien getestet.

Acknowledgments

First of all I would like to thank my advisor, Hans-Jürgen Siegert, for his encouragement, advice, and research support throughout my doctoral studies. I am also deeply indebted to him for giving me the opportunity to lecture on robot motion planning.

I want to thank my current and former colleagues Boris Baginski, Andreas Koller, Oliver Schmid, Gerhard Schrott, and Thomas Weiser. They provided a supportive and most enjoyable working environment. I want especially to thank Boris for introducing me to robot motion planning and for proof-reading this thesis. Sincere thanks go to Jennifer Weeks for correcting my un-English words and phrases.

This thesis would not have been possible without my parents' love and support, as well as my family and friends' encouragement. To all of you, thank you.

Finally, special thanks go to Hans, for carefully reading this thesis, but above all, for his patience, love, and inspiration.

Contents

CHAPTER 1

Introduction and Overview

A short introduction into the significance of robot motion planning is provided. We motivate our work by giving evidence that planning in time-varying environments is useful and important. At the end of this chapter, a preview on the remaining chapters of this thesis is given.

1.1 Introduction

There is no doubt that robots are of great benefit to mankind. Robots assist doctors during medical surgeries, they repeat tedious movements in manufacturing plants, they transport heavy loads, or they observe and control events in hazardous environments. Much progress has been achieved in robotics in the past decades but there are still many open problems that need to be solved in order to reach the ultimate goal of robotics: to create *autonomous robots*. Traditionally, robots are used to perform programmed, repetitious tasks, or tasks where a human operator has to constantly specify the motions. Autonomous robots, however, will accept descriptions of tasks on an abstract level and they will carry out those tasks without human intervention or explicit teaching.

Many different technologies need to be developed in order to reach that goal. Among those are technologies for perception, automated reasoning, planning, manipulation and learning. One of the main planning problems is *motion planning*. Clearly, an autonomous robot must be able to plan its own motions, because by moving in the real world the robot will accomplish its tasks. The classic motion planning problem can be described roughly as follows: given an initial configuration and a final configuration of the robot, find a path connecting both configurations that avoids collisions with obstacles. It is assumed that geometry and position of obstacles is known in advance and it is assumed that obstacles do not move.

In this thesis we consider motion planning in an extended setting. First, we allow obstacles to move and second, we are not only interested in finding a geometric collision-free path but we are interested in finding a path that can be executed by a real-world robot, and to that end we have to take into account robot dynamics such as torque and

force limits.

Consider the following moving obstacle problem. A heating cell is given that has to be loaded and unloaded by a robot. The door to the cell is closed when heating and open otherwise. The robot must plan its motion for loading and unloading respecting the fact that the door is closed from time to time. It would be easy for the robot to wait until the door is fully opened but it might be necessary to start moving the manipulator before the door has even started to open in order to unload and load the cell in time before the door closes again.

To see the necessity for taking robot dynamics into account, consider an athlete lifting a heavy dumb-bell. The athlete does not move the dumb-bell on a straight line. He has to respect the limits of his muscles and he tries to use as little energy as possible to fulfil this task. He will utilise the masses of his forearm and upper arm to get drive. Furthermore, he will spare his back by bending the knees. For a manipulator the same laws of nature are valid and should therefore be taken into account during motion planning.

The trajectory of a manipulator can fulfil various constraints. Some of those constraints must be fulfilled if we want to plan motions for real world, for example, there must be no collisions with obstacles and we must always be within the joint, torque and force limits or we have to ensure that a mobile robot does not overturn (*mandatory constraints*). Other constraints are *optional*, for example, keeping the manipulator some distance away from obstacles for safety reasons or keeping the tool of the robot oriented in a certain way (e.g. to hold a cup filled with liquid). Trajectories can also be evaluated according to certain optimisation criteria. Reasonable optimisations are, for example, to find the time minimal trajectory, the energy minimal trajectory, or a trajectory that minimises the maximum torque and force (in order to spare the robot's mechanics). Our main focus in this thesis will be on techniques for computing trajectories that fulfil all mandatory constraints. In addition, we will give hints on how these techniques can be extended to handle optional constraints and optimisation criteria.

The above mentioned environments can be described as *time-varying* since they are characterised by the presence of bodies that move over time. We assume that the trajectories of all obstacles is known in advance. Typical examples where this assumption holds are manufacturing tasks in which robot manipulators track and retrieve parts from moving band-conveyors or where manipulators have to move between other manipulators. In these environments, the parts on the conveyors and the machine parts of existing robots have a known trajectory. Other time-varying scenarios can be found in aviation and space travel. Consider the problem of navigating a spaceship through a field of asteroids. In this case the trajectories of the asteroids are known in advance. Or think of an object passing a space shuttle where the object has to be caught at a specified time.

The motion planning problem for robots in a known time-varying environment can be described as follows: We are given an initial configuration of the robot and an initial point in time as well as a final configuration of the robot and a final point in time. We are also given the geometries and trajectories of all obstacles. The goal is to find a trajectory that moves the robot from one to the other timed configuration avoiding collision with obstacles and respecting all limits of the robot, including torque and force limits.

Planning in time-varying environments is substantially harder than planning in static environments. In a static environment where all obstacles are fixed, if a safe path from

initial to final configuration has been determined, then a suitable velocity profile can always be found for the robot along that path, provided that the available force and torque enables the robot to at least compensate for gravity in all of its possible positions. In a time-varying environment, however, path planning and velocity planning cannot be performed independently of each other, since the avoidance of moving obstacles depends on the robot's dynamics.

1.2 Overview

This thesis is structured as follows. We start in chapter two by reviewing related work. A classification of path and trajectory planning problems is given accompanied by an overview on known results on the subject. One focus is on results regarding the computational complexity of motion planning in different settings.

In chapter three, we describe our approach to motion planning in time-varying environments. We propose a planner that is able to search in a high dimensional timed search space based on flexible functions for rating the quality of a given trajectory and a set of heuristics for improving trajectory quality, including randomisation to escape local minima.

In the chapter thereafter, we give preliminaries necessary for a formal treatment of motion planning in a time-varying environment. We introduce mathematical definitions related to robots, configuration of robots, trajectories, obstacles and collisions.

One of the novelties in our approach to planning is the distinction between *base point trajectories* and *exact trajectories*. A base point trajectory is defined by a finite set of points and it represents all exact trajectories that run along those points within a certain distance. This distinction enables us to change focus during planning. Sometimes it is necessary to concentrate on overall characteristics of a trajectory and to abstract from details. For example, our heuristics for improving trajectories operate on base point trajectories while the actual rating of quality is done on exact trajectories generated from those base points. In chapter five, the notion of base point trajectories is formalised. Moreover, we analyse different ways how to *generate* exact trajectories from a given base point trajectory, resulting in different well-known types of exact trajectories for manipulators.

The following two chapters, chapters six and seven, are dedicated to the rating of the quality of a given trajectory. Being able to rate the quality of a trajectory in a consistent and reliable way is an important prerequisite for our motion planner. We split rating into two parts. In chapter six, we consider the rating of a trajectory regarding the dynamic limits of the robot. A better rating means less violation of the torque and force limits of the robot. Chapter seven covers the rating of a trajectory regarding collisions of the robot with static obstacles, moving obstacles and its own links. To this end, a novel collision rating is applied that not only detects the absence of collisions but also rates collisions according to their depth. Here, a better rating means less deep collisions or no collisions at all.

In chapter eight, we give a detailed description of our main algorithm for motion planning in time-varying environments. Besides quality ratings, the important part of our motion planning approach is to modify given trajectories in order to find trajectories with

improved ratings. In this chapter, heuristics are given that find alternative trajectories by moving, adding, and deleting base points of a given trajectory.

To demonstrate the usefulness of our approach, we have implemented our motion planning algorithm and we have integrated it into a robot simulation system. In chapter nine, we show results of our algorithm in various planning situations. We analyse the results regarding execution time and trajectory quality for the different parameters of the algorithm. A short description of the robot simulation system which we have developed to test motion planning in time-varying environments can be found there as well.

We conclude in chapter ten by giving a brief summary of this thesis and some remarks on future work.

CHAPTER 2

Related Work

In this chapter we present various motion planning problems and the state of research for algorithms solving these problems for robots. We first classify motion planning using the survey of [Hwang and Ahuja, 1992]. We then move to techniques for modelling the environment and discuss different approaches to testing for collisions. In the third part we take a look at known path planning algorithms, and finally, we focus on trajectory planning approaches in static and time-varying environments.

2.1 Classification of Motion Planning Problems

Motion planning comes in a variety of forms and the simplest version can be described as follows. We are given a robot system, which consists of several rigid objects (links), attached to each other through joints or moving independently, and a three dimensional environment with obstacles. The number of real parameters that determine the robot's links placement are the degrees of freedom of the robot. Each placement is called a configuration of the robot. If there is more than one robot present, then the planning problem is called a *multi-movers* problem. Often it is possible to combine multiple robots to one robot, as this is equal to a single robot with a forked kinematic structure. As for an example, consider two manipulators or a two finger gripper. Furthermore, motion planning is either *constrained* or *unconstrained*. It is called constrained if we have to cope with restrictions due to reasons other than collisions with obstacles. Typical constraint motion planning problems are to keep a cup of tea upright without spilling any liquid during the robot's movement or to stay within the velocity and acceleration limits of the robot. Another class of constraint planning problems is planning for non-holonomic robots, such as car-like robots. Here the curvature of the path of a mobile robot is restricted due to the actual velocity of the robot and the steering capability implied by its wheels.

Depending on the properties of the obstacles in the environment, we can classify the motion planning environment as being either *static*, *time-varying*, or *dynamic*. In a dynamic environment, only partial or no information on obstacles is available at the beginning of the planning process. For example, consider a robot finding its way through

people walking around in a building. In a static environment, obstacles do not move and the location of all obstacles is known in advance. A static environment can often be found in industrial manufacturing where each robot has its own workspace. We speak of a time-varying motion planning problem if the locations or the configurations of obstacles are changing over time but are known before the planning process begins. Imagine already planned movements of another robot or obstacles laying on a conveyer belt. If objects can change their shape, then we say that it is a *conformable* planning problem otherwise it is called *nonconformable*. For example, consider a robot that needs to move flexible cables in a car to reach some configuration behind the cables. If robots can move objects, then it is a *movable-object* problem, e.g. a robot has to open a door or it has to move boxes away to reach the goal.

In this work, we mainly consider manipulators in a time-varying environment where the number of degrees of freedom is equal to the number of joints. The planning problems that we focus on are constrained, since we want to stay within the dynamic limits of the robot. Furthermore, we assume that the obstacles are nonconformable and not movable.

It is advantageous in robot motion planning to distinguish between *physical space* and *configuration space*. The term physical space (or world or Cartesian space) refers to the three dimensional space in which robots and obstacles exist in, at a particular point in time. On the other hand, a time configuration of the world and the robot is the set of independent parameters completely specifying the positions of every point of the obstacles and the robot in the physical space at some point in time. The space of all possible time configurations is called configuration space. If there are no time-varying obstacles, then a configuration is defined without the time, as the time is only used to determine the position of the obstacles. Corresponding to the obstacles in physical space we have C-obstacle regions in the configuration space indicating the configurations resulting in collision (not reachable positions) in the physical space. Each motion planning problem may have its own configuration space and the number of dimensions of the configuration space is usually much higher than three.

In *kinodynamic motion planning* (also called *trajectory planning*) we try to find robot trajectories that simultaneously satisfy kinematic and dynamic constraints. Examples for kinematic constraints are staying within given joint limits or avoiding obstacles. Examples for dynamic constraints are staying within given bounds for velocity and acceleration, or force and torque. This approach is different to the classical architecture (cf. [Latombe, 1996]) which breaks the planning process into three separate parts (see Figure 2.1). First, a continuous collision-free path is generated. In a second step, called trajectory generation, a velocity profile along the path is determined. In a final step, the trajectory is executed. Here, the trajectory is tracked by continuously measuring the robot's actual motion and by computing the forces that need to be exerted by the actuators at each time step in order to perform the desired motion. As observed by Latombe, this process can result in rather inefficient trajectories.

Considerable effort has been put into research on computing trajectories from given collision free and often shortest geometric paths. But apparently, minimal Euclidean length may not be the most suitable criterion for trajectories. A better approach might be to take into account the dynamic equation of the robot in conjunction with the actuators' saturation limits during the planning process [Latombe, 1996]. Moreover, if a problem

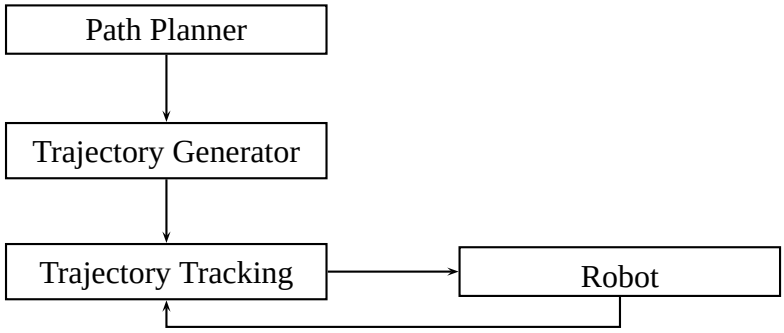

Figure 2.1: *Classical architecture of trajectory planning.*

in a time-varying environment needs to be solved, it is essential to consider the dynamic equations during the planning process and not in a separate step after path planning (although there exists an approach where first a collision free path in the static environment is computed and then, while tracking the path, the velocity is adapted to avoid moving obstacles. We will discuss the limitations of this approach later in this chapter.)

2.2 Geometric Representation and Computation

Because physical objects define spatial distributions in 3D-space, geometric representations and computations play an important role in robotics. Unlike researchers in computational geometry, robotics researchers often pay little attention to the underlying combinatorial and complexity issues. On the other hand, it is sometimes difficult to apply results from computational geometry to robot motion planning, since practical aspects relevant to robotics are often ignored in computational geometry, e.g. only static environments are considered usually. Since testing for collisions between the robot and the environment is essential to motion planning, the representation of geometry is an important issue.

There are four major representation schemata for modelling solids in the physical space [Hoffmann, 1997]. In *constructive solid geometry* (CSG) the objects are represented by unions, intersections, or differences of primitive solids. The *boundary representation* (BRep) defines objects by quilts of vertices, edges, and faces. If the object is decomposed into a set of nonintersecting primitives we speak of *spatial subdivision*. Finally, the *medial surface transformation* is a closure of the locus of the centres of maximal inscribed spheres, and a function giving the minimal distance to the solid boundary. Most solid modelling systems use BRep and there are many methods converting other schemata into BRep [Hoffmann, 1997]. Research has focused on algorithms for computing convex hulls, intersecting convex polygons and polyhedra, intersecting half-spaces, decomposing polygons, and the closest-point problem.

There are two different types of collision tests in connection with robot motion planning: the *static* and the *dynamic collision test* [Stifter, 1991]. The static collision test checks if the robot collides at a specified configuration and point in time with the environment. To make sure that a whole path is collision free, a dynamic collision test is needed.

The dynamic collision test decides for a path or a trajectory if path sections or a time interval is collision free. This means for a given trajectory in a time-varying environment that one has to determine if there is any intersection between the robot and the obstacles during a given time interval in which the robot moves along a given trajectory while the obstacles change their position over time.

Many algorithms for collision detection are based on the use of bounding volumes and techniques for hierarchical spatial decomposition. The goal is to divide the environment in hierarchical sub-groups. Hierarchical spatial data structures are described in [Samet, 1989]. The main difference between various collision tests that use bounding volumes is the shape of the bounding volumes around sub-groups resulting in different ways of finding the sub-groups. The shape of the bounding volumes has to be chosen such that testing for collision between two bounding volumes can be performed efficiently. According to a cost function given by [Weghorst et al., 1984] the choice of the bounding volumes is governed by two conflicting constraints. First, the bounding volume should fit the original model as tightly as possible, and second, testing two such volumes for overlap should be as fast as possible [Gottschalk et al., 1996].

Three different main shapes are commonly used. In [Beckmann et al., 1990] boxes are introduced as bounding volumes with sides parallel to the x-, y-, and z-plane. However, this kind of boxes is not suited for robot motion planning since collision tests have to be done with different orientations of the links of the robot. Recalculation of the hierarchical representation would be necessary each time the robot or an obstacle changes to a different orientation in the world. Another possibility is to use spheres [Hopcroft et al., 1983, Quinlan, 1994, Martínez-Savador et al., 1998]. The third method uses oriented boxes as bounding volumes [Gottschalk et al., 1996]. Here, each box can have an individual orientation in order to make each box fit its enclosed model as tightly as possible. In [Gottschalk et al., 1996], spheres and oriented boxes are compared with respect to the tightness the volumes bound given objects and the number of performed volume tests during many collision tests. The experimental results indicate that one should prefer oriented boxes over spheres.

In [Glavina, 1991] a dynamic collision test is suggested, which enlarges the size of the obstacles by a safety distance and then uses the static collision test to check the path for freedom from collision at discrete steps. The enlargement of the obstacles has to be larger than the maximal possible movement of the robot between two consecutive tests. Later in [Baginski, 1997] this dynamic collision test is improved in three aspects. First, not the obstacles but the robot is expanded. Second, the step size and the expansion size can be modified. This has the advantage that in free areas fewer tests are necessary. Third, the sizes are chosen depending on the position of the link in the kinematic chain, which again reduces the number of tests.

2.3 Basic Path Planning

In this section, we consider the basic path planning problem (also known as the piano mover's problem [Schwartz and Sharir, 1983a]). The problem is to find a geometric collision-free path for a robot in a known static environment. The resulting path is rep-

resented by curve segments each connecting a pair of points in the robot's configuration space [Lozano-Pérez, 1983]. This curve must not intersect the C-obstacle region, which is the image of the obstacles from physical space projected into the configuration space. There has been considerable research on finding collision free shortest paths. [Hwang and Ahuja, 1992, Halperin et al., 1997, Sharir, 1989] give a good survey. In addition we will mention some results on kinodynamic planning in time-varying environments but only for the purpose of comparison. For a detailed presentation of results on kinodynamic planning, we refer to the next section.

Some solutions to the basic path planning problem in static environments can be extended to the problem where we are given a time-varying environment and an unconstrained robot. To this end, the configuration space has to be extended by an additional dimension representing the time [Erdmann and Lozano-Pérez, 1987]. Then ordinary planners for the basic path planning problem can be used to plan in time-varying environments, provided that we can make sure that the planner never decreases the time dimension along a path (because the robot is not allowed to move back in time). However, this approach to plan in time-varying environments fails as soon as dynamic constraints of a robot have to be taken into account because it is likely that in the resulting paths, at some points in time, the force and torque limits of any real robot are exceeded.

Let us now return to static environments. We first focus on complete algorithms and their complexity. A *complete* algorithm is guaranteed to find a collision-free path if one exists, otherwise it returns failure. In [Gupta, 1998] two more subclasses are added. First, the *resolution complete* algorithms, which guarantee to find a collision-free path if one exists at a given resolution. Second, the *probabilistically complete* algorithms that find a collision-free path if one exists with a probability approaching one.

The high complexity of complete path planning methods in high-dimensional configuration spaces makes it necessary to look for heuristic methods that use weaker notions of completeness and that can be partially adapted to specific problem domains in order to boost performance in those domains.

2.3.1 Theoretical Results

The first lower-bound result on the complexity of the piano mover's problem appears in [Reif, 1979]. Reif shows that the mover's problem in a 3D-space is $\mathcal{PSPACE}$-hard even if the mover's problem is generalised to a three-dimensional linkage made of polyhedral links. (The class $\mathcal{PSPACE}$ represents the problems solvable by an algorithm in polynomial space with respect to the input size; a problem is said to be $\mathcal{PSPACE}$-hard, if all other problems in $\mathcal{PSPACE}$ can be reduced to the given problem by a polynomial time transformation [Hopcroft and Ullman, 1979].) It is reasonable to believe that all $\mathcal{PSPACE}$-hard problems are intractable. The result remains true in more specific cases, for example, when the robot is a planar arm in which all joints are revolute [Halperin et al., 1997]. However, it no longer holds in some very simple settings, for instance, planning the path of a planar arm within an empty circle is in $\mathcal{P}$ [Hopcroft et al., 1985]. (The class $\mathcal{P}$ represents the problems solvable by an algorithm in polynomial time with respect to the input size.) In the paper of Reif, a polynomial-space algorithm is presented, proving that the generalised mover's problem is $\mathcal{PSPACE}$-complete [Reif, 1979].

In 1983, Schwartz and Sharir gave another upper bound on path planning complexity [Schwartz and Sharir, 1983b]. They showed that planning a free path in a configuration space with n dimensions and a free space defined by m polynomial constraints of maximal degree d has a time complexity that is exponential in n and polynomial in both n and d. In [Schwartz and Sharir, 1983b] a planning algorithm based on the exact cell decomposition method was given with a double exponential time complexity. Later in [Canny, 1988] a planning algorithm based on a roadmap method with exponential time complexity was proposed. But, as Latombe noted, there exist no implementations of these algorithms that are fast enough to constitute practical solutions [Latombe, 1996].

The first result on the computational complexity of planning with dynamic constraints in time-varying environments has been given by [Reif and Sharir, 1985]. Reif and Sharir analysed the problem where the dynamic limitation of the robot is considered and the obstacles are allowed to move with fixed, known velocity. They showed that planning the movement of a single disk (with velocity constraints) in three dimensions is $\mathcal{P}SPACE$-hard. For the special case of translational obstacle movement (known as asteroid avoidance problem), Reif and Sharir gave an algorithm for the 2-dimensional case (with a 3-dimensional configuration space) that is polynomial if the number of obstacles is constant. Moreover, they gave an exponential time algorithm for 3 dimensions (with a 4-dimensional configuration space). These algorithms run in time $O(n^{2(k+2)}k)$ (2-D) and $O(2^{n^{O(1)}})$ (3-D) respectively, where n is the size of the input (total number of vertices and edges of the obstacles) and k is the number of obstacles.

2.3.2 Complete Algorithms

Two kinds of complete planners have been developed: general ones for an arbitrary number of degrees of freedom and specific ones which apply to restricted families of robots, e.g. robots with a fixed number of degrees of freedom or planar robots. All those planners have in common that they try to divide the configuration space into smaller pieces (allowed and forbidden areas) or to reduce the number of dimensions of the search space (roadmap methods).

We first want to take a closer look at the roadmap method. Here the general idea is to represent the robot's free space in the form of a network of one-dimensional curves (roadmap) lying in the free configuration space. To complete the planning process after the roadmaps are calculated, the start and goal configurations are connected to the roadmap and then a simple graph search in the roadmap is done that finds a path from start to goal.

The roadmap methods, like the visibility graph of Nilsson [Nilsson, 1969], the retraction approach of Ó'Dúnlaing and Yap [Ó'Dúnlaing and Yap, 1982] based on Voronoi graphs, the freeway methods of Brooks [Brooks, 1983], and the general roadmap or silhouette method of Canny [Canny, 1988] are landmarks. However, these methods are either not applicable or very inefficient in higher dimensional problems, as they all try to map the higher dimensional search space to a one or two dimensional space.

The simple visibility graph method [Nilsson, 1969] applies to two-dimensional configuration spaces with polygonal C-obstacles. The roadmap which is called the "visibility graph" is built in the configuration space by connecting every pair of the C-obstacles'

vertices by a straight segment if the segment has no intersection with the interior of a C-obstacle. The method has been mainly used for path planning for mobile robots.

The main idea of the retraction approach [Ó'Dúnlaing and Yap, 1982] is to define a continuous mapping from the configuration space onto a roadmap. In the two-dimensional case with polygonal C-obstacles the simplest choice is to use the Voronoi-diagram. In principle, this method is general, but at the moment mappings exist only for low-dimensional configuration spaces. The freeway method (not complete) in [Brooks, 1983] resembles the retraction approach and was mainly developed for mobile robots, as the method applies to a polygonal robot translating and rotating among polygonal obstacles. The idea is to define a graph (similar to the Voronoi diagram) lying between the obstacles, but in addition giving possible rotation angles on the graph sections.

Finally, the silhouette method [Canny, 1988] solves the basic motion planning problem with a time complexity that is exponential in the dimension of the configuration space. This method presumes that the free configuration space is a compact semi-algebraic set. The idea is to recursively decrease the dimensions of hyperplanes which are swept across the configuration space. Now the roadmap is built by connecting locally extremal points of the free configuration space with piecewise algebraic curves. The recursion then is called in the hyperplanes where the connectivity of the curves changes and hence new extrema appear.

Another extensively studied approach is to decompose the search or configuration space into finite collections of exact cells (cf. [Schwartz and Sharir, 1983a,b, Lozano-Pérez, 1981, Brooks and Lozano-Pérez, 1985]). For this exact cell decomposition, polygons and trapezoids were used in the 2-dimensional case. In the general case of higher dimensions, the decomposition is done in a finite collection of semi-algebraic cells, where the free configuration space again has to be described as a semi-algebraic set. After the partition of the search space into allowed and forbidden cells, a search graph is generated representing the connectivity between the free collection of cells. This graph can be obtained by selecting an arbitrary point in each cell and joining it by a path to the sample point of every cell adjacent to this cell.

All these methods have in common, that they need to calculate the configuration space or the roadmap or the exact cell decomposition explicitly, before any path planning is possible. Many of these methods are limited to configuration spaces of dimension 2 or 3 or require too much effort for pre-calculation, which is often not suitable for planning in changing environments. Nevertheless, some of these methods may be applicable to time-varying environments using the time as an additional dimension of the configuration space. But then the problem is that the trajectories given via the roadmaps may not be traceable by the robot, because of its dynamic limitations. The same problem occurs if one takes the connectivity graph representing the free cells of the configuration space.

2.3.3 Resolution Complete Algorithms

The algorithms of the last subsection work with a mathematically exact representation of configuration space and physical space. Sometimes this is not feasible or an exact representation is not given at all. In contrast to this, resolution complete algorithms work on a discretisation of the configuration space or the physical space. Results on resolution

complete planners can be divided into those that solve general problems with an arbitrary number of degrees of freedom and specific ones which apply to restricted families of robots, e.g., robots with a fixed number of degrees of freedom or planar robots.

The roadmap planners and the exact cell decomposition methods from the last subsection would also fit here, because they use the configuration space for their planning and normally the C-obstacles are not given as polygonal or as semi-algebraic sets in the configuration space, but instead have to be calculated from a given scene in physical space. Often an exact calculation is quite complicated and has to be approximated. Hence, the calculation of the C-obstacles or the free configuration depends on the resolution used for the configuration space calculation. Consequently, these planners turn out to be resolution complete algorithms in practice.

The approximate cell decomposition differs from the exact approach in the fact that the cells are now required to have a simple pre-specified shape, e.g. a rectangular shape. Normally such cells only allow a conservative approximation of the free space. The approximate cell decomposition again splits the configuration space in allowed and forbidden cells. But now the allowed cells are equivalent to free configuration space and the forbidden cells may represent free and/or occupied regions. Here again the search graph is generated representing the connectivity between the free collection of cells.

The approximate cell decomposition approach was first introduced in [Brooks and Lozano-Pérez, 1985]. It was subsequently developed by other researchers, e.g [Laugier and Germain, 1985, Zhang, 1995]. Most cell decomposition methods allow the size of the cells to be locally adapted to the geometry of the C-obstacle region. This is better than keeping the size of the cells fixed, since if the fixed cell size is chosen too large we may not find a path and if the cell size is chosen too small, computation may take too much time.

However, the number of cells to be generated is a polynomial function of the number of semi-algebraic constraints used to model the robot and the obstacles and of the degree of these constraints [Barraquand et al., 1990]. The number of cells also tends to grow exponentially with the number of degrees of freedom. In [Barraquand et al., 1990] it is proposed not to pre-compute the whole connectivity of the free space, but to construct it incrementally, while it is searched. There exist three different methods that follow this approach. In the first two, the resolution of the cell representation is changed hierarchically during the planning process [Duelen and Willnow, 1991, Zhu and Latombe, 1991, Chen and Hwang, 1992, 1996]. In the third method, the calculation of the allowed and forbidden cells is done during the movement from start to goal configuration [Wörn et al., 1998]. Let us take a closer look at each of these methods.

In [Duelen and Willnow, 1991, Zhu and Latombe, 1991] a heuristic for hierarchical path planning is given. Hierarchical path planning is done recursively. The existing cells are labelled as *empty*, *full*, or *mixed*. In each recursion, the actual cell decomposition searches for a free path through *free* cells. If such a path does not exist, another path going through *free* and *mixed* cells is searched. The *mixed* cells are decomposed in smaller cells and then the above is repeated. The heuristic given in [Zhu and Latombe, 1991] tries to maximise the volume of *empty* and *full* cells during the decomposition.

The second method that constructs cells on the fly [Chen and Hwang, 1992, 1996] is called SANDROS (Selective And Non-uniformly Delayed Refinement Of Subgoals) and

has a global and local component. The global component produces a plausible sequence of subgoals (cells) to guide the robot, and a local planner then tests the reachability of each subgoal in the sequence. If the local planner is not successful the global planner has to generate a new sequence of subgoals. For this process it could be necessary to divide cells. This division is only done in one dimension of the configuration space starting with the joint nearest to the base. Now a new subgoal graph is searched, preferring subgoals with large distances to the obstacles. The local planner is quite simple, as it connects two subgoals step by step preferring those steps having a larger distance to the obstacles.

Finally, in [Wörn et al., 1998, Henrich et al., 1998] a parallel online approach with a parallelised A*-search algorithm is proposed. The idea is to map the nodes (voxels) that have to be considered in the next step of the search to the processors. The collision test itself is done by distance calculation. The authors claim that their approach is suitable for time-varying environments, but they do not give a description of the dynamic assumptions and all experiments have been done in static environments. We will compare their results with our own results in Section 9.2.3 (page 138).

2.3.4 Probabilistically Complete Algorithms

The high complexity of path planning for robots with many degrees of freedom has motivated the development of computational schemes that attempt to trade off completeness against time. One such scheme, *probabilistic planning* [Barraquand et al., 1997], avoids computing an explicit geometric representation of the free space.

It samples the configuration space by selecting a large number of configurations at random and retaining only the free configurations as nodes for a connectivity graph (*global planner*). For each pair of nodes a connection is made if there is a collision-free path from one node to the other in the configuration space (*local planner*). These planners are probabilistically complete, that is, if a path exists they will find one with high probability if we let them run long enough.

Various strategies can be applied to sample the configuration space. The strategy in [Kavraki et al., 1996b, Kavraki and Latombe, 1994a,b] proceeds as sketched above. An in-depth analysis can be found in [Kavraki et al., 1996a]. In the first phase, a probabilistic roadmap is generated by sampling the configuration space and connecting the samples by a local planner. They tested various local planners, but finally preferred the fastest and simplest one, which only tries the connection via a straight line. Once a roadmap has been precomputed, it is used to process an arbitrary number of path planning queries. The number of samples generating the roadmap increases the precomputation time. On the other hand, fewer samples will solve less problems. In [Eldracher and Baumann, 1995, Eldracher, 1996] a neural net method has been proposed that adapts the graph to changes in the environment.

The ZZ planner of Glavina [Glavina, 1991] can be seen as a probabilistic planner as well. The idea of this planner is to move on a straight line towards the goal configuration in the configuration space (local planner). This straight line is tested step by step for freedom from collision. Each of these steps can be seen as a new sample in the configuration space. As soon as such a test fails, the strategy of the sample generation changes by taking samples located orthogonal to the moving direction. If these samples can be successfully

reached, the first strategy moving straight towards the goal is used again. A third strategy (global planner) is taken if the sample generation is no longer successful. This may happen if the sample configuration reached before using strategy two is not the nearest configuration of all successfully visited configurations during local planning so far. The third strategy generates random subgoals somewhere in the configuration space and again these subgoals are connected by using the first two sample strategies (local planner). As soon as a connection via subgoals is found the planner stops. The advantage of this planner is that there is no precomputation necessary, as the ”roadmap” is constructed on the fly during the planning process. However, this roadmap is not stored for future planning steps. Since there is no exploration of the full configuration space, this planner is suitable for planning in high-dimensional configuration spaces.

There exist approaches that try to extend the above probabilistic planners to plan in time-varying environments. We discuss these approaches in Section 2.4.3 on page 17.

2.3.5 Heuristic Algorithms

Because of the high complexity of complete planning several heuristic techniques have been proposed to speed up path planning. Heuristic algorithms often search a regular grid defined over the configuration space or they use discrete step sizes and generate a path as a sequence of adjacent grid points and steps respectively. The most impressive results were obtained using potential field methods, where the heuristic function guides the search for a path. Furthermore, the potential field can easily be adapted to the specific problem to be solved, in particular the problem of avoiding obstacles.

For example, Barraquand and Latombe present a potential-guided path planner with random techniques to escape from local minima (RPP - Randomized Path Planner) [Barraquand and Latombe, 1990, Barraquand et al., 1992]. The algorithm starts with the initial configuration executing steepest descent motions in the potential field. If a local minimum is reached it tries to escape with random motions. Each of these motions is immediately followed by a steepest descent motion. For the potential field generation the workspace is discretised into voxels. The potential field is then generated by first computing the distance of all voxels to the obstacles in the workspace using a wavefront algorithm. The robot itself has control points (normally a small number in the order of the number of links) assigned to its links. For each control point of the robot, the distance of the voxels to the control points in goal configurations are computed. Now an arbitrary configuration of the robot can be rated, by using the actual position of the control points in connection with the distance to the obstacles and their individual goal configuration distance. The resulting path is normally quite jerky, and for this reason it is transformed into a smoother path in an extra step at the end.

The BB-method of Baginski can also be seen as a heuristic planner [Baginski, 1996a, 1999]. To our knowledge, it is currently the fastest planner for high-dimensional configuration spaces. This planner has a local heuristic and a global probabilistic component. The global planner is identical to the global planner of Glavina (described above). The local planner works quite differently, as it does not construct a collision free path step by step from start to goal configuration, rather it modifies a whole path using a heuristic controlled by the depth of any occurring collision. This depth rating is done by shrinking

the links of the robot. This is done sequentially until the shrunk robot does not collide any more (if necessary, links get shrunk until they vanish). The amount of shrinking is then taken as the rating of the particular configuration of the robot. By changing a path locally at the section with the worst collision (rating), the rating or potential of the path is changed. The new path is accepted if the rating gets better. The potential itself is given by the portion of the robot which collides. This principle of complete path modification in combination with a rating of collision has been used before, e.g. in [Buckley, 1989].

The advantage of the BB-method and the previously mentioned ZZ planner is, that they are applicable to high dimensional problems, as they do not try to explore the complete configuration space but rather try to modify a given path in the configuration space. The main disadvantage of many heuristical planners using potentials is the presence of local minima in the potential field. Potential field functions that are free of local minima have been proposed, for example in [Rimon and Koditschek, 1992]. However, it is assumed that the computation of these functions is at least as expensive as deterministic and complete path planning itself.

2.4 Trajectory Planning Algorithms

In the next three subsections, we take a closer look at trajectory planning algorithms, that is, algorithms that consider both kinematic and dynamic constraints. There are two main fields of research in the area of kinodynamic motion planning. One approach is to compute a trajectory starting with a given collision free path which is then modified to additionally fulfil dynamic constraints. In this case, path planning is a kinematic problem, involving the computation of a collision-free path from start to goal, whereas velocity planning is inherently a dynamic problem, requiring the consideration of robot dynamic and actuator constraints. The other approach is to compute trajectories directly from scratch respecting kinematic and dynamic constraints simultaneously from the beginning. We focus our attention on motion planners which are able to find collision free trajectories in environments with obstacles (in some work on motion planning an obstacle-free environment is assumed).

First, we want to take a look at two related problems, where the time plays an important role during the planning process. On the one hand, there is *optimal-time control planning* which searches for a time parameterisation of a given path. On the other hand, there is *minimal time trajectory planning*, searching for the fastest trajectory from a start to a goal configuration. Originally neither of these planning problems considered obstacles.

2.4.1 Optimal-time Control Planning

The optimal-time control planning problem is as follows. The input is a geometric collision-free continuous path and the output is supposed to be a velocity profile that lets the robot execute the path as fast as possible. All obstacles are fixed. In this problem it is assumed that the basic path planning problem is already solved. Let $\psi(x)$ be the description of the given path with $0 \leq x \leq 1$. x is the normalised distance from the start configuration ($= \psi(0)$). At the goal configuration ($= \psi(1)$) this distance is one.

The problem is to find the time parameterisation $0 \leq \tau(t) \leq 1$ of $x = \tau(t)$ that minimises the time to travel along $\psi(x)$, while satisfying actuator limits. The equation of motion of a manipulator with m degrees of freedom can be written as $M(c)\ddot{c} + V(\dot{c}, c) + G(c) = \Gamma$, where c,$\dot{c}$, and $\ddot{c}$ respectively, denote the manipulator's configuration, velocity, and acceleration [Craig, 1986]. M is the $m \times m$ inertia matrix of the manipulator, V the m-vector (quadratic in $\dot{c}$) of the centrifugal and Coriolis forces, and G the m-vector of the gravity forces. Γ is the m-vector of the torques applied by the joint actuators.

Research on time-optimal control of robotic manipulator dates back to the early 1970s. Efficient methods for optimising the motion along specified paths have been developed for a rigid manipulator model. One technique developed by [Luh and Walker, 1977, Luh and Lin, 1981] was to minimise the time required to move along a specified path consisting of straight lines and circular arcs. In this work, piecewise constant acceleration and maximum velocity constraints were assumed. Although these assumptions are common in manipulator control, the maximum achievable acceleration and velocities can actually vary substantially with manipulator configuration and angular velocities.

Minimum-time control planning becomes a two-point boundary value problem: Find $\tau(t)$ that minimises t_f, subject to $\Gamma_{\min} \leq \Gamma \leq \Gamma_{\max}$, $\tau(0) = 0$, and $\tau(t_f) = 1$. Numerical techniques solve this problem by doing a discretisation of the path $\psi(x)$ [Bobrow et al., 1985]. Their approach is using the actuators' torques and forces as the control input [Bobrow et al., 1985, Shin and McKay, 1984]. It is assumed that the path of the manipulator's tip is given either with or without orientation and the inverse arm solution can be determined at each point on the path. The basic idea of their solution is to select an acceleration profile that produces the largest velocity profile such that, at each point on the path, the velocity is no greater than the maximum velocity at which the actuators can hold the manipulator on the path. This is done by using so-called switching curves and points. In [Ozaki and Lin, 1996] not the tip of the manipulator but the joint path is taken and optimised by using B-spline curves for each joint.

It was shown that the time-optimal control saturates at least one actuator at all times, and the actuator torque might be discontinuous at the switching points. If the actuator limits represent the true actuator capabilities, the motions that are faster than the time-optimal would obviously result in tracking errors, or deviations from the specified path [Shiller et al., 1996].

This research area is quite important for high productivity in industrial scenarios. There it is desirable that the specified speeds be time-optimal so as to reduce motion time and thus to minimise cycle times. In the classical optimal-time control planning, no constraints other than the dynamic ones are considered. In particular, obstacles are not allowed to move.

2.4.2 Minimal-time Trajectory Planning

Optimal-time control planning finds the fastest trajectory along a given path. However, there might exist paths that allow a shorter travel time.

In the minimal-time trajectory planning problem, the trajectory with the shortest travel time is sought after (again obstacles are assumed to be fixed). Finding a minimal-time trajectory is $\mathcal{NP}$-hard for a point robot under Newtonian mechanics in three dimensional

space [Donald and Xavier, 1995b].

One approach to solve this problem approximately is to find first a geometric free path in the physical space and then to iteratively deform this path to reduce travel time [Shiller and Dubowsky, 1991]. Each iteration requires checking the new path for collision and recomputing the optimal-time control. The algorithm starts by selecting near-optimal paths from the work space grid using a branch and bound search. To allow the search for paths in the physical space, obstacle shadows are defined as regions formed by grid points that are not accessible. These paths are further optimised with local path optimisation to obtain the global optimal solution.

The approximation algorithm in [Donald et al., 1993] and an improved version in [Donald and Xavier, 1995b,a], compute a trajectory ε-close to optimal in time polynomial in both $\left(\frac{1}{\varepsilon}\right)$ and the geometry complexity avoiding static obstacles. The dynamic bounds are given by the maximal velocity and acceleration in each dimension. The algorithm itself plans in the state space. In addition to the dimensions of the configuration space, the state space also contains the velocities in the joints. The search is done in a graph (breadth-first or A^*) discretising the state space. The vertices of the graphs are the states (configuration plus velocity vector) and the edges correspond to a trajectory section (computed by the algorithm) that each takes the same time. The timestep is chosen such that the velocity bound is a multiple of the maximal acceleration multiplied by this timestep. In [Jacobs et al., 1989, Heinzinger et al., 1990] a similar algorithm is presented without consideration of obstacles.

The result in [Yamamoto et al., 1994] and [Mohri et al., 1995] is quite similar to that one given by Shiller and Dubowsky. The collision free minimum time trajectories are generated in two levels. First a collision free path is searched using exact cell decompositions, genetic algorithms or potential fields. These paths are then smoothed by B-splines and finally evolved by a gradient method. The presented experiments were only done in two and three dimensional search spaces with static circular obstacles.

2.4.3 Planning in Time-varying Environments

Only few results on planning in time-varying environments where the dynamic of a robot is considered during the planning process can be found in the literature at the time being. There exist some results for non-holonomic mobile robot motion planning in time-varying environments, but those results are not applicable to higher-dimensional configuration spaces. Results include basic complexity proofs and solutions for restricted cases. In [Lee and Chien, 1987], an overview on time-varying obstacle avoidance for robot manipulators can be found. They classify the approaches into heuristic or analytic approaches, and online or offline approaches. Furthermore, the constraints which have to be satisfied are introduced, e.g. a time constraint (complete the desired motion in the specified period of time), collision constraints, and torque constraints.

In [Kant and Zucker, 1986, 1988], the planning in time-varying environments is split into two sub-problems: the path planning and the velocity planning problem. The path problem consists of computing a path avoiding all static obstacles. The velocity planning problem determines the velocity along that path such that the robot avoids all moving obstacles. This last step is carried out on a space-time plane in which the abscissa is the

arc length of the path and the ordinate is time. Under some circumstances, no trajectory will be found even if one exists. Furthermore, this approach is not suitable for vehicles.

In [Reif and Sharir, 1994], Reif and Sharir investigate the computational complexity of planning the motion of a body in 2-D and 3-D space, so as to avoid collisions with moving obstacles of known, easily computable trajectories. They gave an algorithm for the asteroid avoidance problem where neither the obstacles (with velocity bound) nor the robot (no velocity bounds) may rotate. Their idea is to get from start to goal by two different types of movements: a direct movement, that is, moving the robot with constant velocity until an obstacle is touched, or a contact movement, that is, moving the robot along the boundary of an obstacle.

An extension of the ZZ planner for time-varying environments has been suggested in [Baginski, 1996b]. The Z^3 method applies the ZZ method in the time configuration space. The planner has to ensure not to move back in time during sample generation. No details are given on the used collision test and the used trajectory type, since it is assumed that it can be decided whether the trajectory planned so far is executable by the robot (in the simple case of a robot without dynamic bounds this is always possible). If the trajectory is not traceable, then it is suggested to change the time parameterisation such that the dynamic limits are fulfilled again, but now this 'new' trajectory has to be tested for collisions. The practical results are given for static environments.

In [Fiorini, 1995, Fiorini and Shiller, 1998] an approach is described that uses *velocity obstacles* which define at every point in time the set of colliding velocities between robot and obstacles. The robot and the obstacles are represented as circles and the obstacles are moving translational. This set is computed using the relative velocities between the robot and each obstacle. From this set, the velocities for the robot avoiding the obstacles can be calculated. Within this set the best avoidance manoeuvre is chosen heuristically such that the trajectory resulting from the sequence of manoeuvres reaches the goal and minimises motion time. A simple example is given (2 degrees of freedom), but no computation times are mentioned. In [Shiller et al., 2001], this approach was extended to obstacles with arbitrary trajectories.

In [Kindel et al., 2000], a randomised motion planner for a kinodynamic asteroid avoidance problem is proposed. Here, the obstacles are moving translational with constant velocity. The idea is based on the probabilistic roadmap framework. A sample of the configuration space represents the configuration and the velocity of the robot and in addition the time. A collision free trajectory is generated by expanding a tree consisting of reachable subgoals. The subgoals (or milestones) are generated randomly, preferring regions with few milestones. These milestones are then connected via trajectory sections. The new sample is added to the roadmap if it is admissible. The goal configuration itself has to be randomly chosen within some region, therefore it is necessary, that the sampling algorithm attains the goal region with high probability. Their planner was mainly tested with mobile robots (non-holonomic and air-cushioned).

We would like to close this chapter by remarking that at the time being, and to our knowledge, no kinodynamic motion planners for planning in time-varying environments exist in literature that are able to plan in higher-dimensional search spaces.

CHAPTER 3

An Approach to Planning in Time-varying Environments

We give a rough overview on our approach to motion planning. We sketch our algorithm and discuss its properties.

3.1 Objectives

Our goal is to develop an algorithm that is able to plan motions of robots with many degrees of freedom and forked kinematic structures. Usually the number of degrees of freedom equal the number of joints of a robot. Special cases, where this equality does not hold, include manipulators with grippers or fingers and certain cases of mobile robots. Although we focus our attention on the case where degrees of freedom equals number of joints, our considerations are also valid for the general case.

We attach great importance to the algorithm's ability to handle robots with a large number of degrees of freedom. For example, a standard industry robot used in car production processes has at least six degrees of freedom, and this number easily exceeds 20 in robots specialised for operating in space [Hirzinger et al., 2000]. Hence, our algorithm should be able to search in a high dimensional search space within a reasonable amount of time. Since exact solutions in high dimensional search spaces are very inefficient, we need to find suitable heuristics.

The term *forked kinematic structure* reflects the fact that the kinematic chain of a robot can have branches, that is, the graph of robot link dependency is a tree. Examples of forked kinematic structures are robots with grippers and robots that consist of an arrangement of spatially separated sub-robots.

Our planning algorithm has to find a motion over time for each degree of freedom (i.e. for each dimension in configuration space) starting with a given configuration at a given point in time and ending at a given configuration at another given point in time. This motion must not collide with any obstacle and all dynamic limits of the robot must be kept.

In the following, we first give a rough sketch of our planning algorithm. Then, we take a closer look at each component.

3.2 Outline

The basic outline of our algorithm is not new. Our work has been inspired by recent work on path planning, in particular, the ZZ planner of Glavina and the more powerful BB-method of Baginski [Glavina, 1991, Baginski, 1999]. Both the ZZ and the BB planner can be used to plan in high dimensional search spaces. However, these planners are unable to handle time and hence, they cannot be used to plan in time-varying environments.

Our algorithm for planning motions in time-varying environments proceeds as follows (see Figure 3.1). We start with the trivial base point trajectory consisting of the initial time configuration and the final time configuration. We generate an exact trajectory from this base point trajectory and compute a rating for its quality. Then the following is repeated until a valid trajectory is found (i.e. a collision-free trajectory that respects all kinematic and dynamic constraints of the robot):

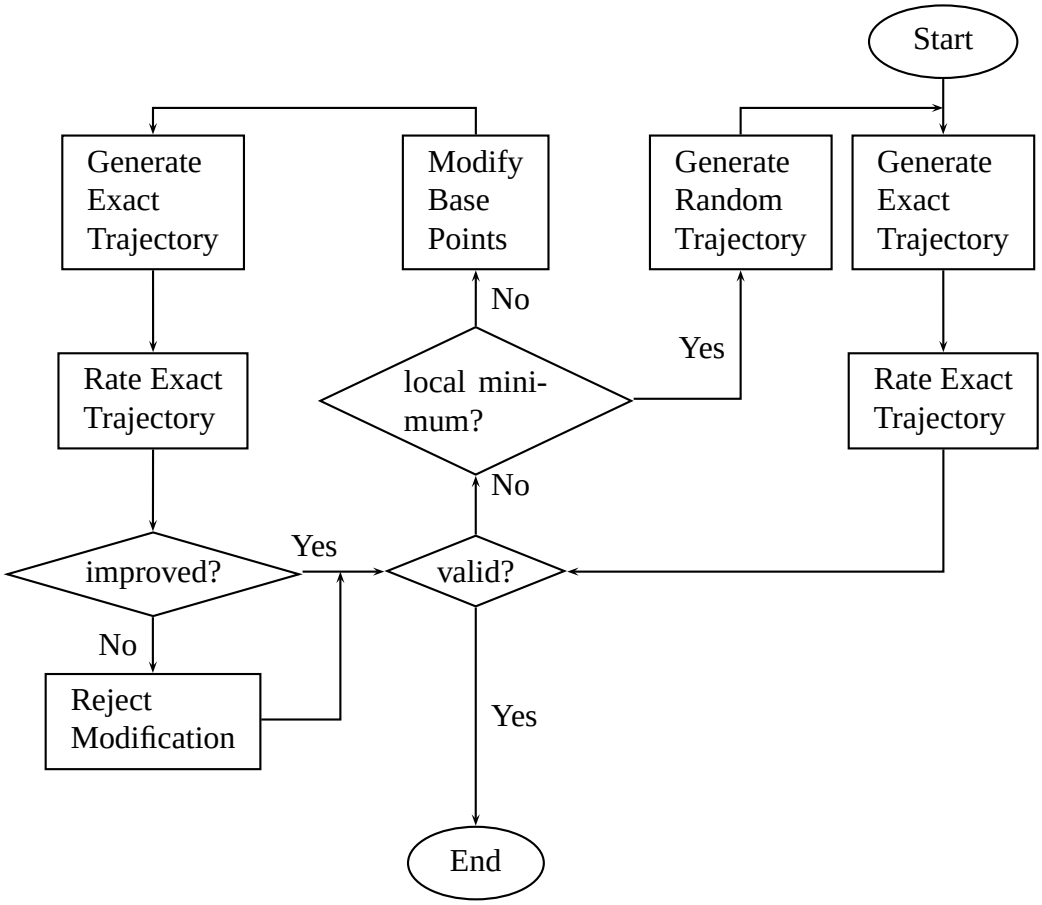

Figure 3.1: *High-level flow chart of the planning process.*

We find and select a piece of the current trajectory with the worst rating. This is

possible since the rating of the trajectory is done piecewise. A piece is a pair of successive base points. We then modify this piece locally until we find a modification that lets us generate a trajectory with a quality rating that is better than the last rating. Possible modifications are moving the base points, inserting a new base point or deleting one of the two base points of the selected piece. If we are unable to improve the rating (because we are stuck in a local minimum), we use randomisation. To this end, we randomly move the piece in configuration space and adjust all neighbouring base points in order to reach a base point trajectory that is likely to fulfil kinematic and dynamic constraints. However, we do not insist any more that the new trajectory has a better rating than the old one. Finally, we start all over by again selecting a piece of the current trajectory with the worst rating and by repeating the above.

A detailed description of this algorithm can be found in Chapter 8. Let us discuss some of its general properties. As we are not able to fully explore configuration space, we try to reach our goal by iteratively improving a given trajectory, that is, by doing a local search. With each step, we try to actually improve except for those occasions where we apply randomisation. When trying to improve, we focus on a single piece of the current trajectory and we always focus on a piece with the worst rating since such a piece is a good candidate for a modification that likely yields a trajectory with a better rating. However, the crucial ingredients of this algorithm are, first, an efficient algorithm that generates exact trajectories from base points, second, a consistent and reliable rating function that reflects collision depth and violation of robot constraints, and finally, a set of fast heuristics to modify a "bad piece" of the current base point trajectory.

Before we take a closer look at those components in the rest of this chapter, let us note that our planner is not *complete*, since we allow arbitrary real valued positions of the base points. Hence, if the planner cannot find a suitable trajectory, then this does not imply that none exists. This shortcoming is inherent to many local planners, e.g. the RPP planner [Barraquand et al., 1992], the ZZ planner [Glavina, 1991], or the BB planner [Baginski, 1999].

3.3 Testing for Collision

To rate trajectories, we need a reliable way of testing whether a given exact trajectory collides with the environment. The collision test should have the following properties. First, if there are collisions, we would like to know how "bad" these collisions are. This qualitative rating will enable our planner to better judge its changes. If our collision test would only return "yes, we have a collision" or "no, we are collision-free", then in many cases, small changes to the trajectory would not change our rating. As a consequence, our planner would be "blind" most of the time. Second, the collision test must be fast since it will be performed many times for each planning task.

A common approach in path planning to obtain a fast collision test, is to check for collision only at certain points of the path. In our case, that means that we check for collisions at certain points in time, for example, we can use the base points from which the given trajectory is generated. To make sure that the robot does not collide between two successive tested points, we need to calculate and respect safety distances. These safety

distances must be chosen such that it is guaranteed that the trajectory does not collide between two tested points in time. This procedure is common to path planning algorithms and variants thereof can be found, for example, in [Glavina, 1991] and [Baginski, 1999]. The critical part of this procedure is that the safety distances must be chosen very carefully. If they are too large, the planner will have difficulties finding its way through areas where the free configuration space is narrow. If they are too small, we might overlook collisions.

In research on path planning, considerable effort has been made to find appropriate safety distance functions. In our case of a time-varying environment, the situation is even more intricate. Our safety distance computation must also take moving obstacles into account. Moreover, while two successive points of a path are always connected by a straight line, two successive points of a trajectory are connected by an arbitrary curve since trajectories contain a velocity profile for the robot. Hence, our safety distance function must cope with curves instead of straight lines. In the following, we give a short overview on our collision test and the safety distance function that we apply. Details can be found in Chapter 7.

Let us first focus on the safety distance. As already mentioned, we test the trajectory for collision at discrete time steps. For each time step, a safety distance between robot and environment is determined. The safety distance will be chosen such that, if no collision is detected at a certain time step, then we know that neither the robot reaches a point outside nor any moving obstacle reaches a point inside this safety distance before the next point of time is reached. As a consequence, if no collision is detected at the discrete time steps, we can be sure that the whole trajectory is collision-free.

In order to reduce the safety distance and to increase precision, we apply two methods. First, at those time steps where a collision has been detected and where the safety distance that we have used exceeds a certain limit, we use bisection to increase precision. We insert a new point in time into the affected time step and repeat the collision test. Second, we use two different kinds of safety distance functions. One is mathematically correct while the other is approximate and is based on heuristics. The approximate function yields smaller safety distances and is correct most of the time. We start planning using only the approximate safety distances. When we have found an “approximately valid” trajectory, we do a collision test using the correct safety distances. If this test fails, we continue planning using the correct safety distances from then on. However, in most cases, the trajectory that our planner finds using the approximate safety distances will already be valid. This approach speeds up the planning process considerably.

Let us now focus on the collision test itself. A well-known idea to obtain a fast collision test is to pre-compute information on the shape of the robot and the obstacles before the planning is started. This information is then reused in every collision test throughout the planning process. The robot and the environment is decomposed hierarchically and for each component on each level simple bounding volumes are determined. The bounding volumes are later used to quickly decide whether components collide or not. A fast method for detecting static collision of arbitrary geometric objects has been developed by Gottschalk, Lin and Manocha [Gottschalk et al., 1996]. Their method uses a hierarchical structure for rapid interference detection where the triangles describing the world and the robot’s links are placed in oriented bounding boxes.

Our collision test is based on the ideas found in [Gottschalk et al., 1996]. In addition, we use a novel approach to account for safety distances and to rate a collision regarding its "depth". This rating is performed as follows. Let s be the sum of all extensions of the robot's links. We test each of the robot's links in the order they appear in the robot's shape starting at the robot's base and ending at the robot's tool(s). Our initial rating is s, and for each link that is free we subtract its extension from our rating. If we find a link in collision, we determine how much of the link is in collision. If p per cent of the link is in collision, we subtract $1 - p/100$ multiplied by its extension from our rating and we ignore all links that follow this link in the robot's topology. As a consequence, the rating ranges from 0 to s, where 0 indicates a free robot and s indicates a full collision. We believe that this rating algorithm adequately reflects collision depth even in the case of a robot with a forked topology.

3.4 Base Point Trajectories

As already mentioned, we distinguish between base point trajectories and exact trajectories. This distinction is one of our keys to planning in time-varying environments. It allows us to switch our focus between modification and evaluation of trajectories. When modifying a trajectory, we operate on base points, that is, we move, add, and delete base points. To rate the quality of a trajectory, however, the base point representation is not sufficient. We need to know the exact position, velocity, and acceleration profile of the robot between each pair of base points.

After each modification of a trajectory, we generate an exact trajectory from the given base points and we do the rating on this generated trajectory. Generation of exact trajectories can be done in various ways resulting in different well-known types of trajectories. In fact, any kind of trajectory generation will do as long as it is guaranteed that when increasing the number of base points along some imaginary path of time configurations, the deviation of the generated trajectory from the imaginary path approaches zero. The idea to prove this for a given trajectory type is to define allowed areas in the configuration space via the base points. Every pair of successive base points defines intervals of allowed values for each degree of freedom and for the time. We have to prove that any generated trajectory lies within these limits. As the number of base points along the imaginary trajectory increases, the allowed intervals get more precise and the planner gets more control over the shape of the exact trajectory that is generated from the base points.

It is also important that exact trajectory generation is fast. In particular, we would prefer trajectory types where a modification of one base point requires only a local modification of an existing exact trajectory. Such a behaviour would suit our planning approach, since our planner performs a local search most of the time.

We have looked at four different types of exact trajectories. Polynomial point-to-point motion, polynomial path motion, a modified bang-bang point-to-point motion, and a modified bang-bang path motion. In the first two motion types we use cubic polynomials to construct the curves between base points, the last two motion types do a linear movement with parabolic blends at the base points. In the point-to-point motion types the robot comes to a full stop at each base point, while in the path motion types the robot can have

an arbitrary velocity at the base points. Each type of motion has certain advantages. Path motions tend to need less torque and force and certainly look smoother. A point-to-point motion can be constructed piecewise for each pair of base points separately, while for a path motion between two base points the neighbouring base points must be considered too. In the modified bang-bang path motion fewer neighbours must be considered than in the polynomial path motion. The modified bang-bang path motion has turned out to be a good compromise between locality and smoothness. See Chapter 5 for a detailed treatment of this issue.

3.5 Trajectory Ratings

To guide the planning process, we need a way to rate the current exact trajectory. Different criteria can be used to base the rating upon. Two of those are mandatory: the trajectory has to be analysed with regard to collisions and the dynamic limits of the robot. Other aspects that may also be considered are overall motion time and energy. All the different ratings are put together to form a single rating for the trajectory and here the ratings can be combined and weighted in various ways to account for differences in magnitude, importance, and characteristic. For example, it is reasonable to ignore the rating of optimisation constraints until the ratings for mandatory constraints indicate that all mandatory constraints are fulfilled. A discussion of this issue can be found in Chapter 8.

The rating itself is a vector of real numbers, whose dimension is the number of sections in the trajectory, that is, each real number in the vector rates a single section. This "piecewise" rating is necessary since our planner needs to know which sections are worse than others.

To rate a trajectory section with respect to collisions, we take the maximum collision depth that we can find for any collision along that section. To rate for robot dynamics, we determine an approximation for the maximum transgression of torque and force limits along the section, accumulated over all joints. A detailed treatment of ratings can be found in Chapter 6 and Chapter 7.

3.6 Improving Trajectory Quality

As already described in an earlier section of this chapter, after rating, the planning process selects a section of the current trajectory with the worst rating. We then try to improve the trajectory by modifying the base points that define this section. If this modification should fail to deliver an improved trajectory, we switch to the next measure: we add a new base point to the section and try to improve the rating by modifying the two new sections. If that fails too, we delete one of the two base points and again try to improve the rating. If that should fail, we do a random movement of the affected section.

In the following, we focus on the strategy used to modify the two base points of a section in order to get an improved rating.

A reasonable strategy in static environments for modifying joints in some colliding configuration is to move the colliding joint orthonormal to some axis of the real world or orthonormal to the direction of robot movement. Such a movement likely results in

a better situation since it directly changes collision depth (cf. [Baginski, 1999]). In our case, where obstacles are moving and collisions are not the only constraint that needs to be observed, things are more complicated. As our rating considers different properties, we cannot assume that an orthonormal movement of the colliding link in physical space improves the rating.

We propose a different, more general method for improving trajectories. Clearly, our ultimate goal is to find the best possible modification in the dimensions of the configuration space. Unfortunately, it is difficult to find an optimal modification since the number of dimensions can be quite large. We therefore use heuristics to find good modifications. One of the heuristics that we have tested works as follows. We try to modify the base points in the robot's configuration space using a certain order on the joints. Initially this order is either top-down or bottom-up along the robot's topology tree. During processing this order will be changed, moving coordinates of the configuration space whose modification resulted in improvements to the beginning of the order. This way, coordinates that have proven to be useful targets for modifications will be the first to be modified in subsequent planning steps. For each degree of freedom in the current order, we look for a better positioning of the two base points in the configuration space by moving orthonormally in the dimensions of the configuration space. After each modification, the trajectory is locally adjusted and rated anew. We repeat this until the two base points cannot be improved further.

As our rating of the collision and the dynamics of the robot is very sensitive, even small changes of a base point get reflected in the rating. This is one of the key properties that enables the above strategy. Experimental results show that this strategy can handle high dimensional robot configuration spaces very well, even with complex movements in the environment and a narrow free configuration space. A detailed description of the planning process can be found in Chapter 8, while experimental results are covered in Chapter 9.

CHAPTER 4

Modelling the World

In this chapter, we introduce terms that we use to describe robots, trajectories, environments, and collisions. A complete listing of symbols can be found on page 171. For further reading on terms and notions used in robotics we refer to [Siegert and Bocionek, 1996] or [Latombe, 1996]. First, we describe the topology of a robot using a directed rooted tree. The next two subsections present definitions regarding the robot's joints and links. Then, definitions for the environment are given and finally trajectories are formally introduced.

4.1 Robots

A robot $\mathcal{R}$ consists of links $\mathcal{L} = \{L_l \mid l = 0,\ldots,n\}$, joints $\mathcal{J} = \{J_j \mid j = 1,\ldots,n\}$, and a quadratic boolean matrix B describing the topology of the robot (b_{ij} is true iff link j follows link i). One joint connects two links. We say that a link L_j is connected to the previous link $L_{p(j)}$ by the joint J_j. $p(j)$ is a function returning the index of the previous link (see 4.3). The number of joints is greater than or equal to the degrees of freedom of the robot. The joints can depend on each other and have no spatial distribution (see Section 4.1.1). The spatial distribution of a link can be zero as well, so we can describe special joints or mobile robots (see Section 4.1.2).

Figure 4.1 shows an example of a robot with six joints. The yellow robot consists of two fingers, which can slide along a rail. Each finger has two revolution joints (J_2, J_3 and J_5, J_6 respectively) and the sliding movement of the finger is via a prismatic joint (J_1 and J_4 respectively). In this case, the robot has *6 degrees of freedom*, or alternatively we say that it is a *6-dof* robot.

The topology of a robot can be described by a rooted directed tree structure, where the nodes are links and the edges are joints (Figure 4.2).

Definition 4.1 Rooted Directed Tree
A rooted directed tree is an acyclic graph. One node is marked as the root. Each edge is

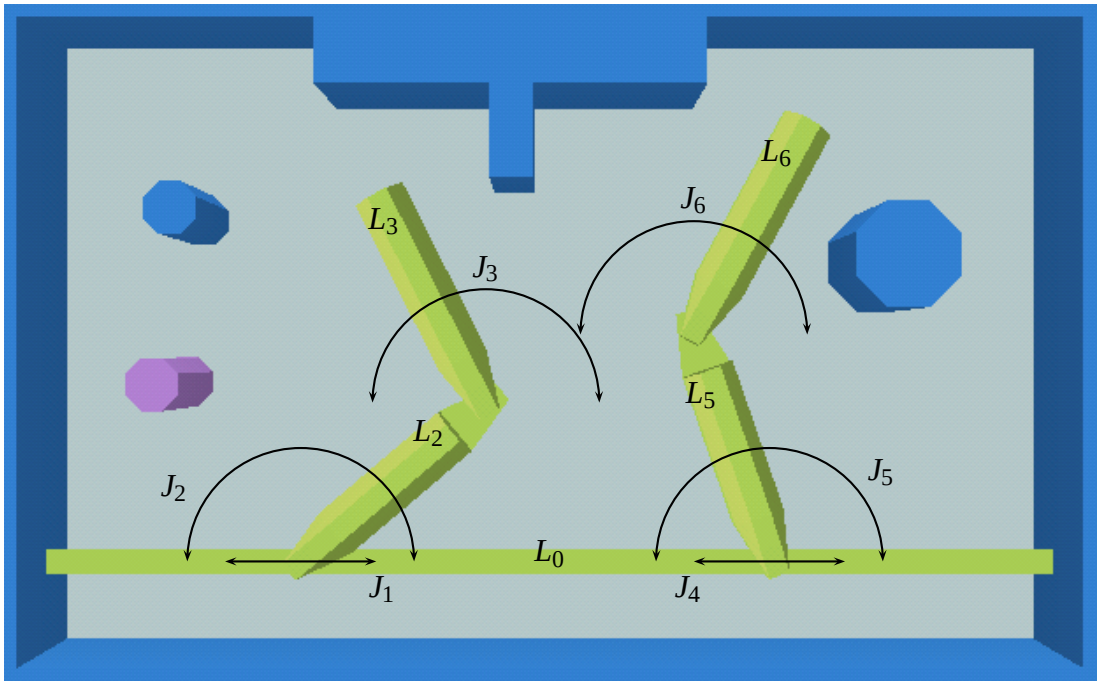

Figure 4.1: *A robot consisting of six joints and seven links (6-dof robot).*

directed. Each node can be reached from the root node. The depth of a node i is equal to the number of edges on the path from the root node to node i.

The root of the tree is the base link L_0 of the robot. The base link L_0 is static in the world and has no previous joint or link. Each edge (joint) connects two nodes (links). The position of a link L_i in the world depends on the previous joints, which are on the path from the base link L_0 to the link L_i. The movement of a joint J_j changes the position of all following joints and links, which have a larger depth and can be reached from the joint J_j. The edges are labelled with a cube, if the connection is via a prismatic joint. The edge of a revolution joint is labelled with a circle.

The directed rooted tree can also be represented as a Boolean matrix B. The dimensions of the matrix are equal to the number of links. If there is an edge from the node L_i to the node L_j, then the value b_{ij} in the i-th row and the j-th column of the matrix B is 1 (true) otherwise it is 0 (false). Our example of the six degrees of freedom robot results in the following Boolean matrix B.

$$B = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{4.1}$$

Now we want to look at the transitive closure of a Boolean matrix. To this end, we

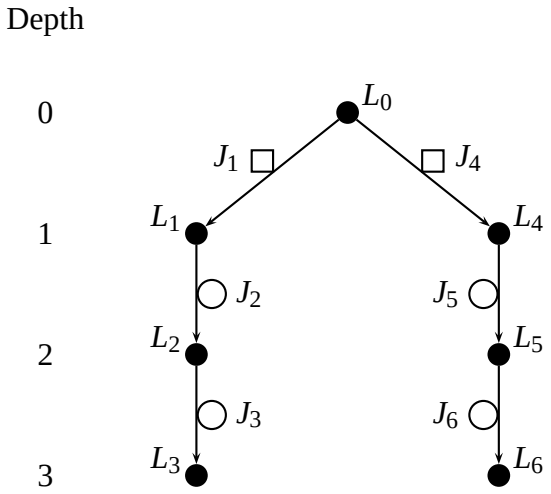

Figure 4.2: *The rooted directed tree of the six degree of freedom robot in Figure 4.1*

need definitions for a Boolean product, a Boolean power, and a Boolean union for Boolean matrices.

Definition 4.2 Product of Boolean Matrices
Let A and B be Boolean matrices of dimension $n \times n$. Then the Boolean product $C = A \wedge B$ is defined as

$$c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj}$$

Hence, a connection between i and j via k exists (in C), if there is a connection from i to k (in A) and a connection from k to j (in B).

Definition 4.3 Power of a Boolean Matrix
Let A be a Boolean matrix of dimension $n \times n$. Then the Boolean power A^i is defined as

$$A^i = \bigwedge_{k=1}^{i} A = \underbrace{A \wedge \cdots \wedge A}_{i \text{ times}}$$

Since the Boolean matrix of a directed robot tree represents the reachability from one link via exactly one joint to another link, the i-th power of a matrix represents the reachability from one link via i joints to another link.

Definition 4.4 Union of Boolean Matrices
Let A and B be Boolean matrices of dimension $n \times n$. Then the Boolean union $C = A \vee B$ is defined as

$$c_{ij} = a_{ij} \vee b_{ij}$$

This means that a connection between i and j exists in C, if there is a connection from i to j in A or a connection from i to j in B.

Definition 4.5 Transitive Closure of a Boolean Matrix
Let I be the Boolean $n \times n$ identity matrix and let A be a Boolean matrix of dimension $n \times n$. Then the transitive closure A^* of A is given by

$$A^* = I \vee A^1 \vee A^2 \ldots A^{n-1}$$

The transitive closure gives us the reachability between links via an arbitrary number of joints.

In our six degrees of freedom robot example (see Figure 4.1) the transitive closure of B is

$$B^* = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.2}$$

We will use the transitive closure in the next two subsections to define sets for previous and next joints and links.

4.1.1 Joints

A joint J_j is the connection of the link L_i to the link L_j, where $p(j) = i$. The function $p(j)$ returns the index of the previous joint of J_j. Let b_{ij} be the elements of the Boolean matrix B, which represents the topology of a given robot. Then

$$p(j) = \begin{cases} -1 & : \text{ if } \forall i, b_{ji} = 0 \\ i & : \text{ if } b_{ji} = 1 \end{cases} \tag{4.3}$$

The spatial distribution of each joint is zero. The connection between two links is either via a revolute or a prismatic joint. In Figure 4.3, a revolution joint (marked with a circle) and a prismatic joint (marked with a box) are depicted.

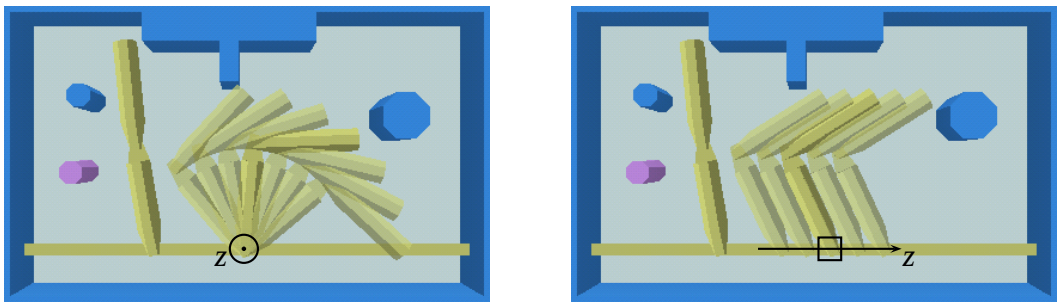

Figure 4.3: *Example of a revolution joint (left) and a prismatic joint (right).*

Each joint J_j has its own coordinate system. The location of this coordinate system is given by a homogeneous 4×4 matrix ${}^0D_j$. This matrix gives the orientation ${}^0R_j$ and position ${}^0p_j$ of the joint's origin relative to the world's coordinate system.

$$ {}^0D_j = \begin{pmatrix} & {}^0R_j & & {}^0p_j \\ 0 & 0 & 0 & 1 \end{pmatrix} = ({}^jD_0)^{(-1)} \tag{4.4} $$

If a homogeneous point ${}^jp$ is given in the joint's coordinate system, then the point's position in the world is ${}^0D_j\ {}^jp$. The movement of a joint is either around or along the z-axis of the joint's coordinate system. Therefore the transformation matrices ${}^0D_j$ change their values depending on the joint's values. To simplify our formulae, we introduce an indicator λ_j that determines the type of the j-th joint.

$$ \lambda_j = \begin{cases} 1 & : \quad \text{if the } j\text{-th joint is a revolute joint} \\ 0 & : \quad \text{if the } j\text{-th joint is a prismatic joint} \end{cases} \tag{4.5} $$

The coordinate system of the joint J_j in zero position is in a relative position ${}^iD_j$ to the coordinate system of the previous joint J_i, where $i = p(j)$ (see Figure 4.4). Furthermore, the relative position depends on the position υ_j of the joint J_j itself. The value υ_j of the joint J_j gives the rotation of a revolution joint around the z-axis or the translation of a prismatic joint along the z-axis of the joint's coordinate system. $\upsilon_j^{\min}$ and $\upsilon_j^{\max}$ are the minimal and maximal possible values for υ_j, $\upsilon_j^{\min} \le \upsilon_j \le \upsilon_j^{\max}$.

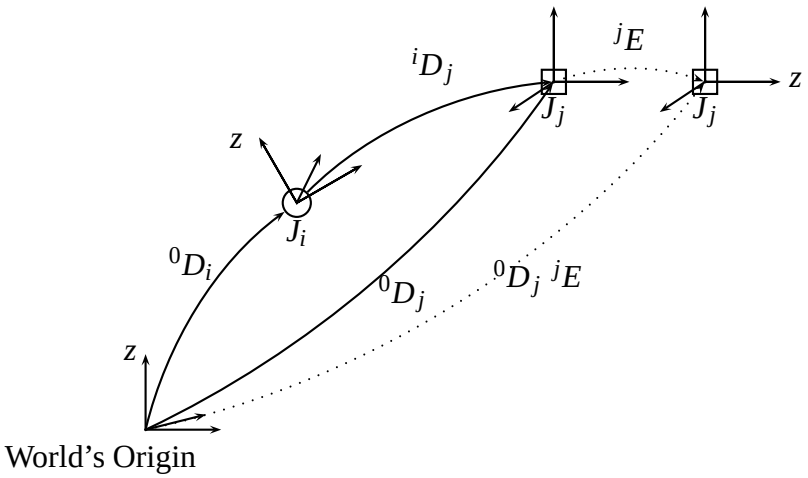

Figure 4.4: *Relative position of two joints.*

The transformation matrix ${}^0D_j$ from the coordinate system of link L_j in the world can be calculated as follows, provided that the transformation matrix ${}^iD_j$ of the previous link L_i and the actual position ${}^0D_i$ of the link L_i is known.

$$ {}^0D_j = {}^0D_i\ {}^iD_j\ {}^jE(\upsilon_j) = {}^0D_{p(j)}\ {}^{p(j)}D_j\ {}^jE(\upsilon_j) $$

The matrix ${}^iD_j$ is equal to the Denavit-Hartenberg notation [Denavit and Hartenberg, 1955, Siegert and Bocionek, 1996] of a joint in zero position, which means that ${}^iD_j$ is the transformation matrix if $\upsilon_j = 0$. The value of the current joint υ_j is not considered during calculation of the following joint positions (${}^0D_l, p(l) = j$), but in the transformation matrix ${}^0D_j$ of joint J_j. This means that the coordinate systems of joint J_j and link L_j are identical.

For a revolution joint, ${}^jE(\upsilon_i)$ is given by a matrix describing a movement around the *z*-axis,

$${}^jE(\upsilon_i) = \begin{pmatrix} cos(\upsilon_i) & -sin(\upsilon_i) & 0 & 0 \\ sin(\upsilon_i) & cos(\upsilon_i) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and for a prismatic joint, ${}^jE(\upsilon_i)$ describes a movement along the *z*-axis.

$${}^jE(\upsilon_i) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \upsilon_i \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

If all joints are independent, then the n independent υ_j values give the exact position and orientation of each joint and therefore a unique position of the whole robot, which is called the n-dimensional configuration of the robot. If there are dependent joints (see Figure 4.5) like a gripper, a soft gripper joint [Hirose, 1993], or a helicoidal joint [Husty et al., 1997], then the dimension of the configuration is given by the number of independent joints. This number is equal to the degrees of freedom of the robot.

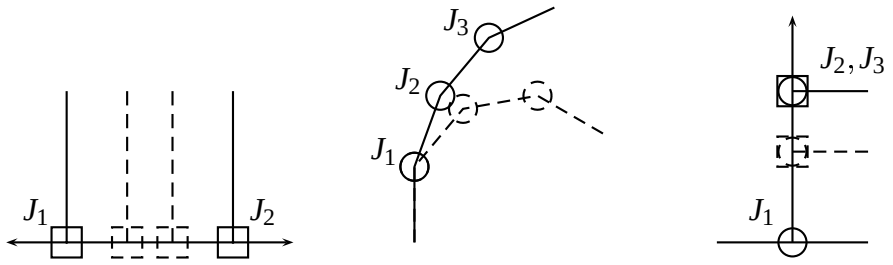

Figure 4.5: *Examples of dependent joints: gripper (left), soft gripper (middle), helicoidal joint (right)*

The gripper on the left side moves its shells synchronously in the opposite directions of the same distance. The joints of a soft gripper are moved synchronously at the same angle. The helicoidal joint is similar to a bolt and nut. Screwing the bolt and not screwing the nut results in an up and down movement of the nut. Therefore this joint consists of two rotational joints (J_1 and J_2), which have the opposite angle value. The translation value (J_3) depends on the angle value.

Definition 4.6 Dependency Functions of a Robot
Let a robot $\mathcal{R}$ consist of n joints J_j and let m be the number of dimensions of the robot's configuration space. The function $d : \mathbb{N} \to \mathbb{N}$ is called the index dependency function of the robot if d maps the index of a joint to the dimension of the configuration space that the joint depends on. It holds that

$$d(j) = i, \text{ for } 1 \leq j \leq n \text{ and } \forall i, 1 \leq i \leq m : \exists j : d(j) = i$$

In addition, there are n value dependency functions $f_j : \mathbb{R} \to \mathbb{R}$, one for each joint J_j, that map the value $\omega_{d(j)}$ of the configuration space dimension $d(j)$ to the value of the depending joint j:

$$f_j(\omega_{d(j)}) = \upsilon_j, \text{ for } 1 \leq j \leq n$$

For a robot without depending joints the index dependency function and the value dependency functions are usually defined as $d(j) = j$ and $\forall j : f_j(x) = x$. But in the helicoidal joint case in Figure 4.5, the index dependency function is $d(j) = 1$ and the value dependency functions are for example $f_1(x) = x$, $f_2(x) = -x$, and $f_3(x) = x$. The configuration of a robot clearly indicates the position of all joints in the world.

Furthermore we assume that, for all joints J_j with the same index $d(j)$, the range of the depending function that defines the valid positions for the joint J_j is continuous. Moreover, for each dimension i of the configuration space, there exists a lower bound ω_i^l and an upper bound ω_i^u such that

$$\begin{array}{lll} \upsilon_j^{\min} \leq f_j(\omega_{d(j)}) \leq \upsilon_j^{\max} & : & \text{if } \omega_{d(j)}^l \leq \omega_{d(j)} \leq \omega_{d(j)}^u \\ (f_j(\omega_{d(j)}) < \upsilon_j^{\min}) \vee (f_j(\omega_{d(j)}) > \upsilon_j^{\max}) & : & \text{if } (\omega_{d(j)} < \omega_{d(j)}^l) \vee (\omega_{d(j)} > \omega_{d(j)}^u) \end{array}$$

Definition 4.7 Configuration of a Robot
Let robot R consist of n joints and let d and f_j be the dependency functions of the robot. Moreover, let the image of d be the set $\{1, \ldots, m\}$. Then a configuration of the robot is an m-dimensional vector $\vec{\omega}$. The set of all valid configurations form the configuration space $\mathcal{CS}$ of the robot. A configuration is valid if

$$\forall j : \upsilon_j^{\min} \leq f_j(\omega_{d(j)}) \leq \upsilon_j^{\max}$$

The vector $\vec{\upsilon} = (f_1(\omega_{d(1)}), \ldots, f_n(\omega_{d(n)}))$ is the joint configuration of the real robot in the world.

Apart from the joint limits $\upsilon_j^{\min}, \upsilon_j^{\max}$, the dynamic limits of the joints are given. Let $\theta_j^{\max}$ denote the maximal torque of joint J_j if $\lambda_j = 1$ or the maximal force if $\lambda_j = 0$.

In the following, we give definitions for sets concerning a joint. The set of previous joints $\mathcal{J}_j^{\mathbf{pj}}$ of a joint J_j consists of all joints, which can modify the position of J_j including J_j. These are exactly the edges on the path from the root to link L_j. Hence, the cardinality of $\mathcal{J}_j^{\mathbf{pj}}$ is the same as the depth of node L_j in the robot's topology tree.

Definition 4.8 Set of Previous Joints of a Joint
Let B be the rooted tree matrix of a robot. Then the set of previous joints $\mathcal{J}_j^{\mathbf{pj}}$ of a joint J_j is

$$\mathcal{J}_j^{\mathbf{pj}} = \left\{ J_i \mid (B^*)_{ij} = 1 \wedge i \neq 0 \right\}$$

All positions of the joints in the set of next joints $\mathcal{J}_j^{\mathbf{nj}}$ of J_j can be modified by J_j including J_j. In the topology tree these are the edges to all successors of J_j.

Definition 4.9 Set of Next Joints of a Joint
Let B be the topology tree matrix of a robot. Then the set of next joints $\mathcal{J}_j^{\mathbf{nj}}$ of a joint J_j is

$$\mathcal{J}_j^{\mathbf{nj}} = \left\{ J_i \mid (B^*)_{ji} = 1 \right\}$$

Finally, we are interested in all next links of a joint J_i. These are all links, which change their position in the world, if joint J_i is moved. These are all links belonging to the joints in the next joint set $\mathcal{J}_j^{\mathbf{nj}}$.

Definition 4.10 Set of Next Links of a Joint
Let B be the rooted tree matrix of a robot. Then the set of next links $\mathcal{J}_j^{\mathbf{nl}}$ of a joint J_j is

$$\mathcal{J}_j^{\mathbf{nl}} = \left\{ L_i \mid (B^*)_{ji} = 1 \right\}$$

After giving definitions for previous and next joint sets, we have to look at the transitive closure B^* in connection with depending joints. If depending joints exist, then the interpretations that a joint changes its position if a previous joint moves and that all next joints change their positions if the joint moves itself do not hold. For example, take the soft gripper in Figure 4.5 and let the index dependency function be

$$d(j) = \begin{cases} 1 & : \text{ if } j = 1,3 \\ 2 & : \text{ if } j = 2 \end{cases}$$

Then the transitive closure for the soft gripper is

$$B^* = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}^* = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

This means for the definition of the set of previous joints that the joint coordinate system of joint J_2 moves, if J_1 or J_2 moves, but this is not correct, as J_2 moves as well if J_3 changes its position.

We would like to modify the transitive closure in the sense that it should not only represent the robot's topology but also the dependencies of joint movement, that is, we would like to know which joint moves if some other joint is changed. As all dependent joints J_i of one joint J_j change the joint J_j itself, the dependent joints J_i have to change at least the same joints as joint J_j. We can give a modified transitive closure definition as follows.

Definition 4.11 Modified Transitive Closure
Let B^* be the transitive closure of the robot's topology matrix B and let d be the index dependency function. Then the modified transitive closure $\hat{B}^*$ is

$$(\hat{B}^*)_{ij} = \bigvee_{d(i)=d(l)}^{l} (B^*)_{lj}$$

Considering again the previous soft gripper example, the modified transitive closure is

$$\hat{B}^* = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}^{\hat{}\,*} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}.$$

Let $\hat{\jmath}_j^{\mathbf{pj}}$, $\hat{\jmath}_j^{\mathbf{nj}}$, and $\hat{\jmath}_j^{\mathbf{nl}}$ be the respective sets belonging to the modified transitive closure $\hat{B}^*$. Using the modified transitive closure we can say, the coordinate system of joint J_j changes its position in the world, if any joint $J_i \in \hat{\jmath}_j^{\mathbf{pj}}$ is moved. On the other hand, all elements in $\hat{\jmath}_j^{\mathbf{nj}} \cup \hat{\jmath}_j^{\mathbf{nl}}$ change their position if joint J_j is modified.

The complete definition of a joint J_j is given by a tuple consisting of the transformation matrices ${}^iD_j$, the minimal and maximal position values and the maximal torque value ($\upsilon_j^{\min}$, $\upsilon_j^{\max}$, $\theta_j^{\max}$), the index of the configuration dimension $d(j)$ that J_j depends on, and the dependency functions $f_j(\omega_{d(j)})$.

4.1.2 Links

A link L_i is a rigid physical body modelled as a compact manifold with boundary. The boundary of the links are given via facets. The facets of the links are relative to the coordinate system of joint J_i. The links $L_i, 1 \leq i \leq n$ are connected via the joints J_i to $L_{p(i)}$ and therefore the link L_i is moving with the joint J_i. Each robot has one base link L_0, which is the root of the robot tree. The base link has no previous joint and no previous link and is static in the world. The position of link L_i depends on the positions of the previous joints. For L_0 the set of previous joints $\mathcal{L}_0^{pj}$ is empty. If the joints of the robot are independent, then the set of previous joints contains all edges of the rooted tree on the path from the root to node L_i and the cardinality of $\mathcal{L}_i^{pj}$ is the same as the depth of the node in the tree.

Definition 4.12 Set of Previous Joints of a Link
The set of previous joints $\mathcal{L}_i^{pj}$ of a link L_i is

$$\mathcal{L}_i^{\mathbf{pj}} = \left\{ J_j \mid (B^*)_{ji} = 1 \wedge j \neq 0 \right\}$$

Therefore we can say that the position of link L_i in the world depends on joints $J_j \in \mathcal{L}_i^{pj}$. Let $\hat{\mathcal{L}}_i^{pj}$ be the set of previous joints according to the modified transitive closure $\hat{B}^*$. Hence, the link L_i changes its position if any joint $J_j \in \hat{\mathcal{L}}_i^{pj}$ is modified.

The mass m_i of the link is in the centre of gravity of the link. The position of the centre of gravity relative to the coordinate system of L_i and J_i is g_i. I_i is the 3×3 inertia tensor matrix of link L_i with respect to its centre of gravity. The inertia tensor is a symmetric matrix, containing the moments and products of inertia.

Since it is possible that a link has no spatial distribution, we can model special joints and robots. For example, in Figure 4.5 the second link L_2 of the helicoidal joint has no spatial distribution. Another example is a mobile robot (see Figure 4.6).

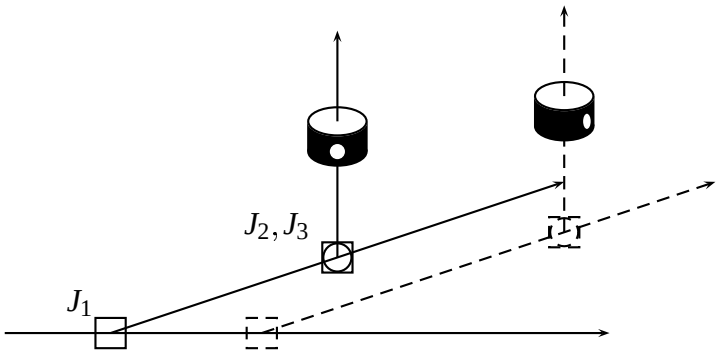

Figure 4.6: *A mobile robot modelled with three joints.*

The possible position of the mobile robot in x and y direction on a plane can be realized with two prismatic joints. Furthermore, a revolute joint can be used to define the orientation of the robot. So the helicoidal joint and the mobile robot have the same joint combination, but the difference is that the joints of the mobile robot are independent.

The complete definition of a link is given by the boundary (a set of facets), the mass m_i, the centre of gravity g_i, and the inertia tensor I_i. For reasons of simplicity, if a triangle or facet t is element of the link L_i then we say $t \in L_i$.

4.2 Environment

The environment consists of obstacles $O = \{O_i \mid i = 0, \ldots, o\}$, which can move over time. If an obstacle does not move, then we say that it is a *static obstacle* otherwise it is a *time-varying obstacle*. The boundaries of the obstacles are modelled with facets relative to the coordinate system of the obstacle. For the obstacles we write $t \in O_i$ if a triangle or facet t is element of the obstacle O_i.

For each obstacle O_i, there is a function $o_i : \mathbb{R} \to \mathbb{R}^6$, returning the orientation and position $o_i(t)$ of the obstacle's coordinate system at time t. Let $o_i(t)$ be $(o_{i,1}(t), o_{i,2}(t), o_{i,3}(t), o_{i,4}(t), o_{i,5}(t), o_{i,6}(t))^T$ in the world at time t. The first three values of the vector return the position in x, y and z direction and the last three values represent the rotation amount around the x, y, and z axis of the world (yaw-pitch-roll system [Siegert and Bocionek, 1996]). If the obstacle is static, then the translation matrix $D_{O_i}(t)$ is constant over time. Otherwise the obstacle coordinate system $D_{O_i}(t)$ at time t is positioned as follows

(for simplicity of writing we drop the parameter t).

$$D_{O_i}(t) = \begin{pmatrix} m_{11} & m_{12} & m_{13} & o_{i,1} \\ m_{21} & m_{22} & m_{23} & o_{i,2} \\ m_{31} & m_{32} & m_{33} & o_{i,3} \\ 0 & 0 & 0 & 1 \end{pmatrix}, \text{ with}$$

$$\begin{aligned}
m_{11} &= \cos(o_{i,4})\cos(o_{i,5}) \\
m_{12} &= \cos(o_{i,4})\sin(o_{i,5})\sin(o_{i,6}) - \sin(o_{i,4})\cos(o_{i,6}) \\
m_{13} &= \cos(o_{i,4})\sin(o_{i,5})\cos(o_{i,6}) + \sin(o_{i,4})\sin(o_{i,6}) \\
m_{21} &= \sin(o_{i,4})\cos(o_{i,5}) \\
m_{22} &= \sin(o_{i,4})\sin(o_{i,5})\sin(o_{i,6}) + \cos(o_{i,4})\cos(o_{i,6}) \\
m_{23} &= \sin(o_{i,4})\sin(o_{i,5})\cos(o_{i,6}) - \cos(o_{i,4})\sin(o_{i,6}) \\
m_{21} &= -\sin(o_{i,5}) \\
m_{22} &= \cos(o_{i,5})\sin(o_{i,6}) \\
m_{23} &= \cos(o_{i,5})\cos(o_{i,6})
\end{aligned}$$

In Figure 4.7, a lilac time-varying obstacle is moving over time. The obstacle's position is changing over time according to the function

$$o_i(t) = (4t, 30\sin(0.1t), 0, 0, 0, 0)^T$$

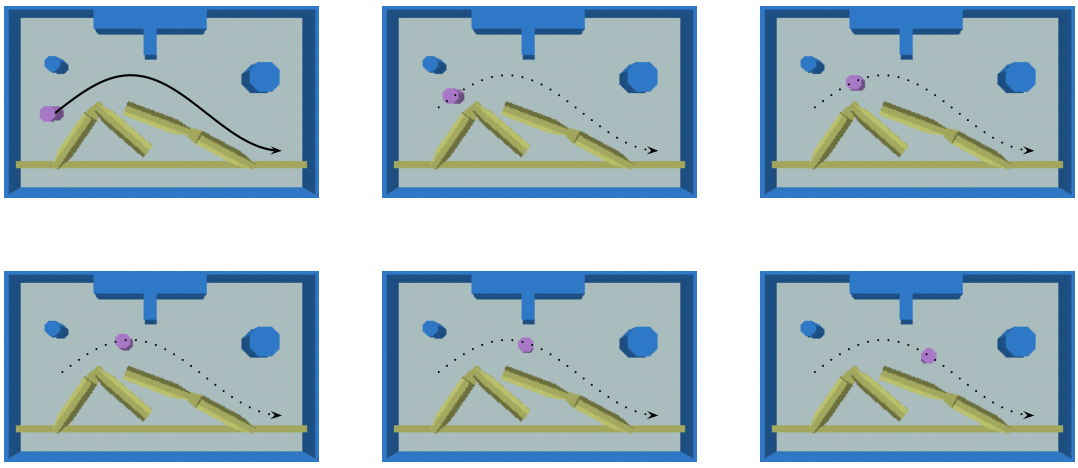

Figure 4.7: *Movement of a time-varying obstacle shown in intervals of five seconds.*

In addition to the function $o_i(t)$, the maximal speed values of each obstacle are given. The speed values of the time-varying obstacles are needed, because we can only look at the environment at discrete time steps. Between these time steps we have to estimate the position of the obstacle. This is again a six dimensional vector $\vec{\sigma}_{O_i} = (\sigma_{O_i,1}, \sigma_{O_i,2}, \sigma_{O_i,3}, \sigma_{O_i,4}, \sigma_{O_i,5}, \sigma_{O_i,6})^T$. The first three values give the absolute value of the maximal speed in

the translational directions of the axis and the last three in the rotational directions around the axis. The maximal speed vector $\vec{\sigma}_{O_i}$ specifies the maximal speed in the obstacle coordinate system centre, which is equal to $\vec{o}_{O_i}$.

In the example in Figure 4.7, the maximal speed value $\vec{\sigma}_{O_i}$ is

$$\vec{\sigma}_{O_i} = (4,3,0,0,0,0)^T$$

Since a configuration determines the exact position of all joints of a robot, we need to add the time to a configuration to fix the positions of the obstacles as well.

Definition 4.13 Time Configuration of the World
Let $\vec{\omega}$ be a valid configuration (see Definition 4.7) of a robot R and let t be a specific point in time. Then $\vec{\vartheta} = (\vec{\omega}^T, t)^T$ is called a time configuration. The configuration space at time t contains all time configurations at time t and is denoted by $\mathcal{CS}_t$.

A time configuration determines the position of links (and joints) of the robot and of all static and time-varying obstacles at a certain point in time.

4.3 Collisions

In Section 4.1.1 we have introduced the configuration and the configuration space of a robot. In Section 4.2 we have extended this to a time configuration. In this section, we introduce collisions and define colliding and free configurations.

Definition 4.14 Colliding and Free Configuration
A time configuration $\vec{\vartheta}$ is called a free configuration, if there is no intersection of the boundary of the obstacles with the links' facets of the robot and if there is no intersection of the boundary of link L_i with the boundary of link L_j, for $i \neq j$. If a configuration is not free, then it is a colliding configuration.

Hence, each time configuration is either colliding or free. In the rest of this thesis, we always mean a time configuration even if we speak of a configuration. All free configurations constitute the set of free configurations $\mathcal{CS}_f$ and all colliding configurations form the set of colliding configuration $\mathcal{CS}_c$.

Definition 4.15 Colliding and Free Configuration Space
The set of all free configurations is called free configuration space $\mathcal{CS}_f$ and the set of all colliding configurations is called colliding configuration space $\mathcal{CS}_c$. It holds that

$$\begin{aligned} \mathcal{CS}_f \cup \mathcal{CS}_c &= \mathcal{CS}_t \\ \mathcal{CS}_f \cap \mathcal{CS}_c &= \emptyset \end{aligned}$$

In Figure 4.8 and Figure 4.9, a cut through the configuration space of the six degrees of freedom robot example from Figure 4.1 can be seen.

Figure 4.8 shows the configuration space in the middle and eight example configurations around it. The dependency functions for all joints are $d(j) = j$ and $f_j(x) = x$. The

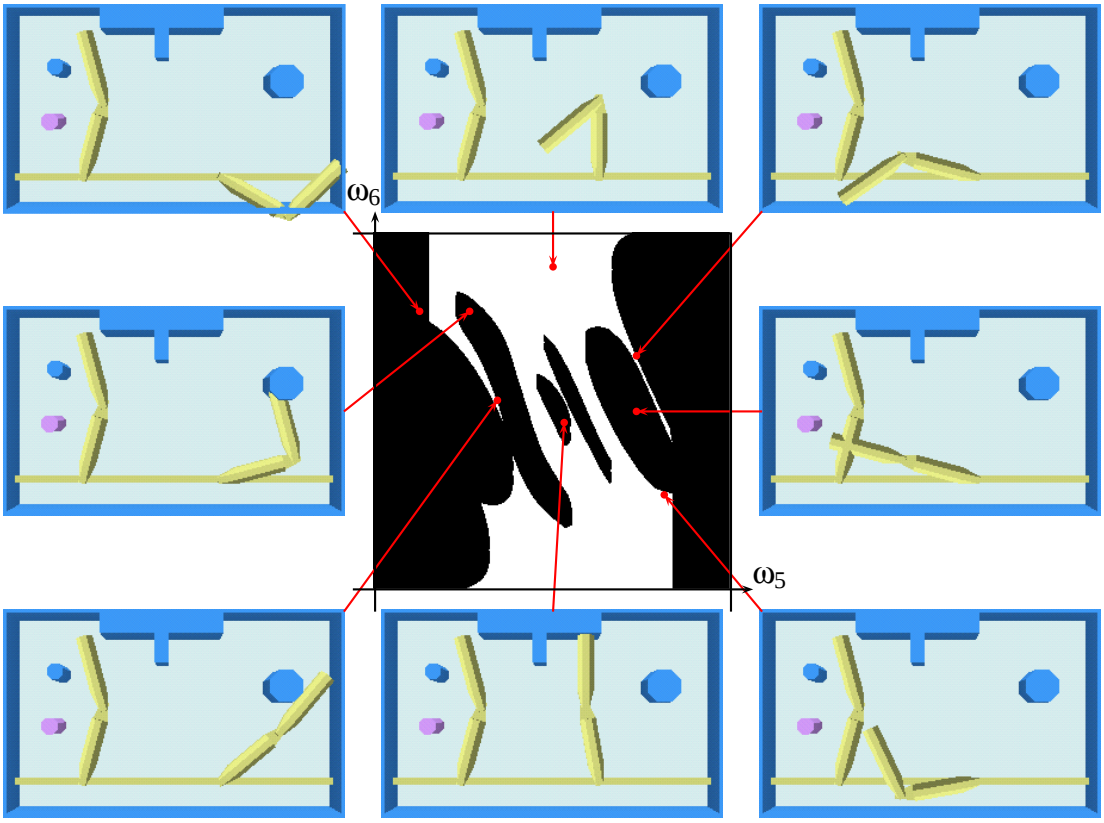

Figure 4.8: *Two dimensions of the configuration space of a 6-dof robot. Here the dimensions for the joint J_5 and the joint J_6 are shown.*

black area in the configuration space indicates colliding configurations and the white area represents free configurations. The values of the joints J_1, J_2, J_3, and J_4 are constant. The value of joint J_5 is placed on the horizontal axis while the value of J_6 is placed on the vertical axis. The values for both joints range between -160 and 160 degrees. The time is kept constant, so the lilac obstacle does not move. The red arrows are pointing from the time configuration in the real world towards the respective point in the configuration space.

The next Figure 4.9 shows a different cut through the same configuration space. This time, the values of the joints J_1, J_2, J_3, J_4, and J_5 are kept constant and J_1, J_2, J_3, and J_4 have the same values as before. Time is drawn along the horizontal axis and the value of J_6 is placed on the vertical axis. The time ranges between 0 and 50 seconds and the range of joint J_6 is again between -160 and 160 degrees.

4.4 Trajectories

In this section, we formally introduce trajectories. Trajectories are used to describe the movement of a robot in the world. Our trajectory does not directly describe a movement

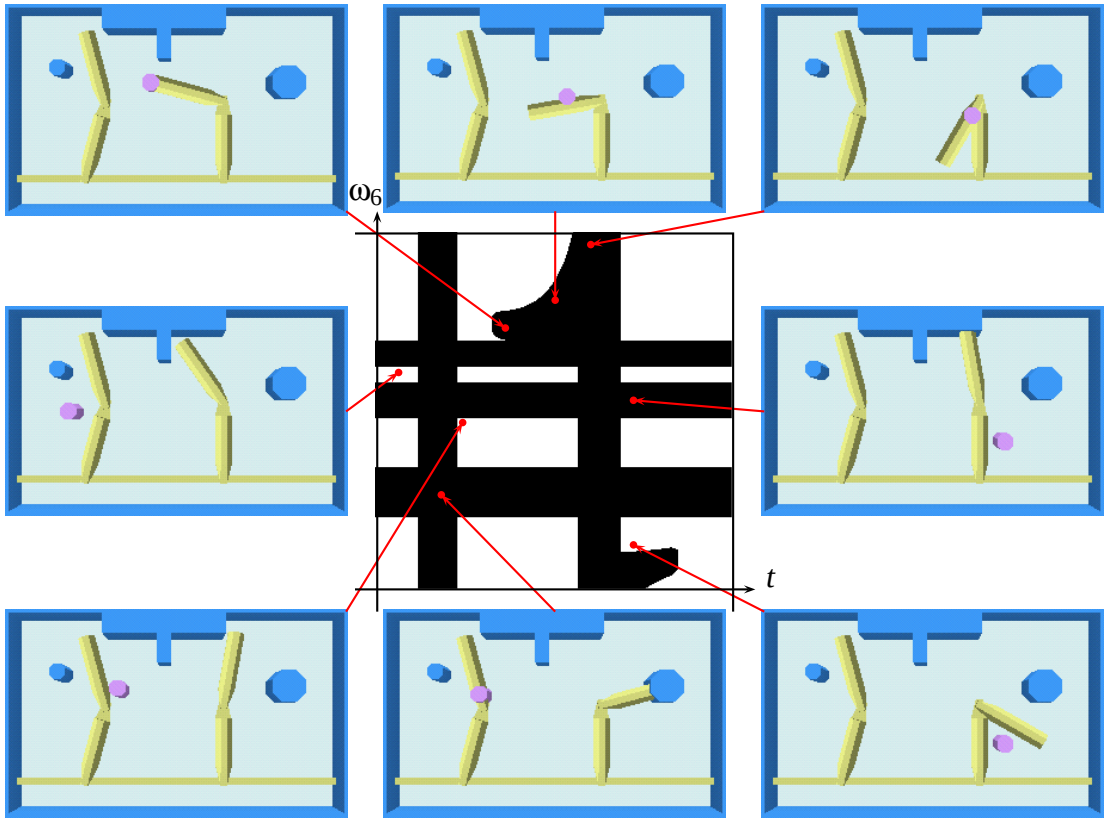

Figure 4.9: *Two dimensions of the configuration space of a 6-dof robot. Here the dimensions for the time and joint J_6 are shown.*

in the real 3-D world, but rather in the configuration space, as we do our planning in the configuration space.

The trajectory is a function of time that returns a value in the dimension m of the configuration space or the degrees of freedom of the robot (without the time). The function (4.6) returns a valid configuration $\vec{\omega}^t$ of the robot at a given time t. The function (4.7) is the trajectory for a single dimension i in the configuration space.

$$T : \mathbb{R} \rightarrow \mathbb{R}^m \qquad (4.6)$$
$$T(t) = \vec{\omega}^t$$
$$T_i : \mathbb{R} \rightarrow \mathbb{R} \qquad (4.7)$$
$$T_i(t) = \omega_i^t$$

$T_i(t)$ has to be continuously differentiable (see [Kerne et al., 1988] for definition of differentiable) for $1 \leq i \leq m$ (4.9), since the derivation gives the velocity in the configurations. Therefore the first derivation (4.8) returns a vector $\vec{\sigma}^t$ representing the speed at time t for

all dimensions of the configuration space.

$$\dot{T} : \mathbb{R} \to \mathbb{R}^m \tag{4.8}$$
$$\dot{T}(t) = \left(\frac{T_1(t)}{dt}, \ldots, \frac{T_n(t)}{dt} \right)^T = \vec{\sigma}^t$$
$$\dot{T}_i : \mathbb{R} \to \mathbb{R} \tag{4.9}$$
$$\dot{T}_i(t) = \frac{T_i(t)}{dt} = \sigma_i^t$$

For a real robot, it is important that the velocity is continuous in each joint. The values $\omega_i, 1 \leq i \leq m$ of the configuration space are not necessarily equal to the real joint values $\upsilon_i, 1 \leq i \leq n$ of the robot since the joint values are calculated via the dependency functions $f_j(\omega_{d(j)})$ (see Definition 4.6). Therefore the dependency functions $f_j(\omega_{d(j)})$ must be continuously differentiable as well, if the velocity in each joint needs to be continuous.

$\dot{T}(t)$ itself has to be derivable in each dimension as well (4.10). The second derivation $\ddot{T}_i(t)$ represents the acceleration at time t for each configuration dimension i (4.11).

$$\ddot{T} : \mathbb{R} \to \mathbb{R}^m \tag{4.10}$$
$$\ddot{T}(t) = \left(\frac{\dot{T}_1(t)}{dt}, \ldots, \frac{\dot{T}_n(t)}{dt} \right)^T = \vec{\alpha}^t$$
$$\ddot{T}_i : \mathbb{R} \to \mathbb{R} \tag{4.11}$$
$$\ddot{T}_i(t) = \frac{\dot{T}_i(t)}{dt} = \alpha_i^t$$

It is sufficient to have differentiability since it is not necessary to get a continuous path for the acceleration of a joint.

Given a robot and a trajectory, it is possible to calculate the position, the velocity, and the acceleration in each modelled joint of the robot at a specific time. The modelled joints are not necessarily equal to the real joints, as we have demonstrated in the examples of the helicoidal joint (Figure 4.5, page 32) or the mobile robot (Figure 4.6, page 36).

Definition 4.16 Dynamic Configuration
Let a robot, the dependency functions (see Definition 4.6), and a trajectory T be given. Then $\vec{\upsilon}_d$, the dynamic configuration at time t (also denoted by $T^D(t)$), consists of the position vector $\vec{\upsilon}_\omega = \vec{\upsilon}$, the speed vector $\vec{\upsilon}_\sigma$, and the acceleration vector $\vec{\upsilon}_\alpha$ of the modelled joints:

$$\begin{aligned}
\vec{\upsilon}_d &= \left((\vec{\upsilon}_\omega)^T, (\vec{\upsilon}_\sigma)^T, (\vec{\upsilon}_\alpha)^T \right)^T = T^D(t) \\
\upsilon_{\omega,j} &= f_j(T_{d(j)}(t)) = f_j(\omega^t_{d(j)}) \\
\upsilon_{\sigma,j} &= \dot{f}_j(T_{d(j)}(t)) \dot{T}_{d(j)}(t) \\
\upsilon_{\alpha,j} &= \ddot{f}_j(T_{d(j)}(t)) (\dot{T}_{d(j)}(t))^2 + \dot{f}_j(T_{d(j)}(t)) \ddot{T}_{d(j)}(t)
\end{aligned}$$

If limitations for the modelled joints in speed or acceleration are given as well, then it is possible to determine if the trajectory is traceable. The dynamic configuration $\vec{\upsilon}_d$

has to satisfy the limitations of the joints at any time t. But as already mentioned in Section 4.1.1, only the maximal torque of each joint is known.

The next step is to take the dynamic configuration $\vec{\upsilon}_d$ and a dynamic model of the robot to calculate the torques in the real joints. For a manipulator consisting of simple revolution joints and prismatic joints and all joints modelled separately with exactly one joint, a very simple model is given by $M(\vec{\upsilon}_\omega)\vec{\upsilon}_\alpha + V(\vec{\upsilon}_\sigma, \vec{\upsilon}_\omega) + G(\vec{\upsilon}_\omega)$ (cf. Section 2.4.1 on page 15). M is the $n \times n$ inertia matrix of the manipulator, V is the n-vector of the centrifugal and Coriolis forces, and G is the n-vector of the gravity forces. We give an exact calculation of the forces and torques for forked manipulators in Chapter 6. We say that a trajectory T is *valid* if the dynamic limits are fulfilled at each point in time.

It is more involved to get the resulting torque for a helicoidal joint or a mobile robot. In the helicoidal joint case, the three values of the joints modelling this special joint are combined to calculate the resulting torque in the joint's drive. For a mobile robot it is different as the values are not connected to a joint, but may lead to an overturn of the mobile robot or cannot be satisfied at all. For example, a mobile robot cannot move against the direction of its wheels. However a definition of the modelled joint's torque limits has to be given. An appropriate model is needed, that gives us the resulting torques according to the robot's situation. Hence, we need a function $M : \mathbb{R}^{3n} \to \mathbb{R}^n$ witch takes the dynamic configuration $\vec{\upsilon}_d$ at time t and returns the force or torque values $\vec{\tau}$ for each joint.

4.5 The Motion Planning Problem

In this section, we use the definitions of the previous sections to formalise the trajectory planning problem in time-varying environments, which is to find a valid trajectory between two time configurations that is collision free at any time t.

The given problem consists of one robot $\mathcal{R}$. The robot $\mathcal{R}$ itself is composed of $n+1$ links $L_l, l = 0, \ldots, n$ and n joints $J_j, j = 1, \ldots, n$. The boundary of the links is given via facets. Furthermore the mass m_l, the centre of gravity g_l, relative to the coordinate system of joint J_l, and the inertia tensor matrix I_l are given for each link L_l. The joints are defined with their relative position ${}^{p(j)}D_j$ to the coordinate system of the previous joint, and the maximal joint values $(\upsilon_j^{\min}, \upsilon_j^{\max})$ and torque value $(\theta_j^{\max})$. The topology and dependencies of the robot is given by a matrix B and the dependency functions (namely the index dependency function $d(j)$ and the value dependency functions $f_j(x)$). Furthermore, we need a dynamic model $M : \mathbb{R}^{3n} \to \mathbb{R}^n$ of the robot.

The environment is given in the form of obstacles $O_i, i = 0, \ldots, o$. The boundaries of obstacles are described using facets. For each obstacle, the position D_{O_i} of the obstacle is given, depending on time t. In addition, a maximal speed vector $\vec{\sigma}_{O_i}$ is provided for each obstacle.

For a given time configuration $\vec{\vartheta}_s = (\vec{\omega}_s^T, t_s)^T$ (start) and a second time configuration

$\vec{\vartheta}_g = (\vec{\omega}_g^T, t_g)^T$ (goal) a trajectory T has to be found such that

$$\begin{aligned}
T(t) &= \vec{\omega}_s^T \text{ for } t \leq t_s \\
T(t) &= \vec{\omega}_g^T \text{ for } t \geq t_g \\
\forall t \text{ with } t_s \leq t \leq t_g &: (T(t), t) \text{ is a free configuration i.e. } (T(t), t) \in \mathcal{CS}_f \\
\forall t \text{ with } t_s \leq t \leq t_g &: \text{the dynamic configuration } \vec{\upsilon}_d = T^D(t) \text{ satisfies the} \\
&\quad \text{dynamic limits i.e. } \forall j = 1, \ldots, n : \mid M(T^D(t))_j \mid \leq \theta_j^{\max}
\end{aligned}$$

Obvious conditions are that the start time has to be smaller then the goal time $t_s \leq t_g$. The first two equations indicate, that the robot stands still in the start configuration and comes to a complete stop in the goal configuration. Moreover, we presume that the start configuration and the goal configuration are collision-free.

CHAPTER 5

Generating Trajectories from Base Points

We first give requirements for describing exact trajectories using base points. We then show how exact trajectories can be generated using two well know trajectory types for point-to-point and path motions. We prove that these types fulfil our requirements. Finally, we give an outlook on other trajectory types.

5.1 Introduction

A trajectory is a function of time in the dimensions of the configurations space (see Section 4.4). As there are different types of trajectories, like point-to-point motions or path motions, we decided to use two levels of trajectory descriptions. The upper level (base point trajectory) gives an approximate run of the trajectory via base points and is the subject of planning. The lower level (exact trajectory) gives the exact run of the trajectory. Depending on the given base points and the kind of trajectory, the exact trajectory over time can be generated (see Figure 5.1). The advantage of this strategy is that the planning can be done with different kinds of trajectories without modifying the planning process. As there are two levels, we need requirements for the connection between the two levels and properties of the exact trajectory that enable us to successfully plan the robot's motion.

The planning algorithm itself needs operations that enable it to change the approximate trajectory and consequently the exact trajectory. There are two basic operations.

- *Adding* a base point at time t. This operation first looks for a position i such that the $(i-1)$-th base point has a smaller time and the i-th base point has a larger time. Then the base point is inserted between these two base points. If the times of all base points are larger or no base point exists, then the new base point is inserted as the first base point. On the other hand, if all times are smaller, then the base point is appended at the end.

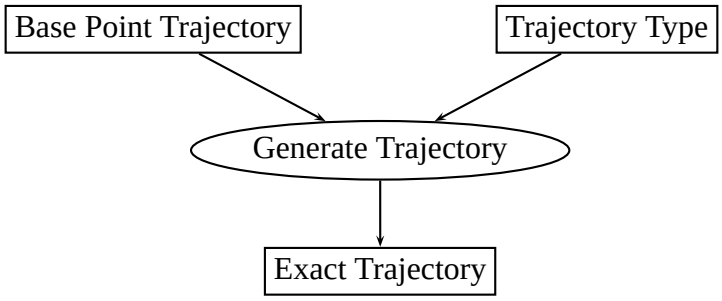

Figure 5.1: *Chart of the trajectory generation process.*

- *Deleting* the base point number i. When a base point is deleted, nothing else changes.

With these two operations, base points can also be moved, since this is the same as deleting the base point which has to be moved and inserting it with a new time and a different position. For both basic operations the exact trajectory has to be recalculated in the vicinity of the change. The size of the vicinity that requires recalculation depends on the type of the exact trajectory that we are using. We will discuss this issue in the next sections, where we introduce examples of trajectory types.

To ensure progress in the planning process and to get reasonable results, we give requirements which have to be considered for the definition of a trajectory via base points and the generation of an exact trajectory.

- *Trajectory Generation* - The trajectory generated from the base points has to be derivable twice in all dimensions of the configuration space (see Section 4.4, page 39). Hence, the trajectory itself is a continuous function over time in the configuration space. Furthermore, the derivations have to be continuous as well. This results in continuous position and speed values over time for each dimension in the configuration space. Another important point for fast motion planning is the local stability of the generation. This means that a small change to a base point should not result in a large change of the exact trajectory.

- *Trajectory Type* - The planning algorithm should handle different types of trajectories, like point-to-point or path motions. Even trajectories for mobile robots should be possible. The planning algorithm uses the upper level of the trajectory description (the base point description) and is therefore independent of the type of the trajectory.

- *Trajectory Exactitude* - As more base points are used to describe the trajectory, the exact trajectory has to converge towards the imaginary trajectory lying on the base points. This means that the allowed area for the exact trajectory has to decrease as more base points are inserted on that imaginary trajectory. In addition, the impact of moving a base point increases when more base points are added.

- *Robot Limits* - The allowed area for the exact trajectory should respect the joint limits of the robot. However, it does not need to respect the robot's dynamic limits, like speed or acceleration. This is the task of the trajectory planner.

In the next section, we give a definition for base point trajectories that fulfils the requirements regarding *trajectory exactitude* and *robot limits*. This is followed by examples of well-known trajectory types where we show that the requirements regarding *trajectory generation* and *trajectory type* are fulfilled.

5.2 Base Point Trajectories

In the following, we give a definition for describing the trajectory $T(t)$ via base points, which ensures *trajectory exactitude* and *robot limits* of the above list. A trajectory is defined via $m+1$ base points $\vec{b}_0,\ldots,\vec{b}_m$. A base point is a time configuration in the configuration space $\vec{b}_i = \vec{\vartheta}_i = (\vec{\omega}_i, t_i) = T(t_i)$, where the n dimensions of $\vec{\omega}_i$ are the degrees of freedom of the robot representing the position values (see Subsection 4.1.1, page 30)

$$\vec{\omega}_i = (\omega_{i,1},\ldots,\omega_{i,n}).$$

For a pair of base points $\vec{b}_i, \vec{b}_l$ it holds that $t_i < t_l$ if $i < l$, for all $i,l = 0,\ldots,m$. The trajectory section $S_i, 0 < i \leq m$ is the trajectory $T(t)$ between the base points $\vec{b}_{i-1}$ and $\vec{b}_i$ covering the time $]t_{i-1},t_i]$. The section S_0 consists of the trajectory at time t_0 (which is a single point). Since we will use base points to define an allowed area for the trajectory run, we need two distance values for each trajectory section.

First, we allow that the exact trajectory misses each base point by a certain amount. Assume that for each base point, there exist $2n$ *epsilon values*

$$\vec{\varepsilon}_i = \left(\varepsilon_{i,1}^{\min},\ldots,\varepsilon_{i,n}^{\min},\varepsilon_{i,1}^{\max},\ldots,\varepsilon_{i,n}^{\max}\right), \varepsilon_{i,j}^{\min},\varepsilon_{i,j}^{\max} \geq 0$$

that give the maximal allowed distance by which an exact trajectory may miss the values $\omega_{i,j}$ at time t_i. A suitable choice for these epsilon values will be given later. For each base point $\vec{b}_i$ with configuration $\vec{\omega}$ at time t_i and the trajectory $T_j(t)$ in dimension j the following holds

$$\omega_{i,j} - \varepsilon_{i,j}^{\min} \leq T_j(t_i) \leq \omega_{i,j} + \varepsilon_{i,j}^{\max}. \tag{5.1}$$

To solve the planning problem and to reach the start from the goal configuration, it is necessary that the epsilon values $\varepsilon_{i,j}^{\min}, \varepsilon_{i,j}^{\max}$ are zero for the first base point ($i = 0$) and the last base point ($i = m$).

Secondly, we define allowed areas between each pair of successive base points that an exact trajectory must stay within. A base point $\vec{b}_i$ defines an allowed area for the run of a trajectory in section S_i. To this end, we assume that for each base point there exist $2n$ *delta values*

$$\vec{\delta}_i = \left(\delta_{i,1}^{\min},\ldots,\delta_{i,n}^{\min},\delta_{i,1}^{\max},\ldots,\delta_{i,n}^{\max}\right), \delta_{i,j}^{\min},\delta_{i,j}^{\max} \geq 0$$

that give a maximal allowed distance between the values of the exact trajectory section $S_i, 0 < i \leq m$ and the values $\vec{\omega}$ in the time interval $]t_{i-1}, t_i[$. For all configurations $T(t)$ between two successive base points $\vec{b}_{i-1}$ and $\vec{b}_i$ the following must hold

$$\omega_{i,j} - \delta_{i,j}^{\min} \leq T_j(t) \leq \omega_{i,j} + \delta_{i,j}^{\max}, t_{i-1} < t < t_i. \tag{5.2}$$

Clearly, the delta values $\delta_{0,j}^{\min}, \delta_{0,j}^{\max}$ must be zero for the first base point (actually they are not used at all, since the trajectory section before the first base point is not considered). To enable a successful trajectory generation, it is necessary that the epsilon and delta values are chosen such that there are no gaps between allowed trajectory areas. This implies that there has to be at least one configuration lying in the delta area of $\vec{b}_i$, in the epsilon area of $\vec{b}_i$, and in the delta area of $\vec{b}_{i+1}$.

We can include the epsilon and delta values into the base points and we speak of *extended base points* $\vec{b}'_i, 0 \leq i \leq m$ that define allowed areas for an exact trajectory:

$$\vec{b}'_i = \left(\vec{\omega}_i, \vec{\varepsilon}_i, \vec{\delta}_i, t_i\right)^T, \text{ with } i = 0, \ldots, m.$$

Let us now turn to the choice of the epsilon and delta values. If the epsilon and delta values can be arbitrary they would have to be considered in the planning process. This would increase the complexity of planning. To simplify planning and furthermore to fulfil the requirement of *trajectory exactitude*, we choose the epsilon and delta values of the base point $\vec{b}_i$ according to its previous and next section, since the base point $\vec{b}_i$ describes the section connecting S_{i-1} and S_{i+1}. In the following, we define suitable epsilon and delta values. Let $\omega_{i,j} = \omega_{0,j}$ if $i < 0$ and let $\omega_{i,j} = \omega_{m,j}$ if $i > m$.

$$\begin{aligned}
\delta_{i,j}^{\min} &= \omega_{i,j} - \min\left(\omega_{i-2,j}, \omega_{i-1,j}, \omega_{i,j}, \omega_{i+1,j}\right) \\
\delta_{i,j}^{\max} &= \max\left(\omega_{i-2,j}, \omega_{i-1,j}, \omega_{i,j}, \omega_{i+1,j}\right) - \omega_{i,j} \\
\varepsilon_{i,j}^{\min} &= \begin{cases} 0 & : \text{ if } i = 0, m \\ \omega_{i,j} - \max\left(\omega_{i,j} - \delta_{i,j}^{\min}, \omega_{i+1,j} - \delta_{i+1,j}^{\min}\right) & : \text{ if } 0 < i < m \end{cases} \\
\varepsilon_{i,j}^{\max} &= \begin{cases} 0 & : \text{ if } i = 0, m \\ \min\left(\omega_{i,j} + \delta_{i,j}^{\max}, \omega_{i+1,j} + \delta_{i+1,j}^{\max}\right) - \omega_{i,j} & : \text{ if } 0 < i < m \end{cases}
\end{aligned} \tag{5.3}$$

This choice ensures that the allowed area gets smaller when more base points are added on an imaginary trajectory. On the other hand, we get no gaps between consecutive sections.

In Figure 5.2 the base point trajectory is defined via six base points $\vec{b}_0 = (2,2)^T$, $\vec{b}_1 = (6,6)^T$, $\vec{b}_2 = (10,11)^T$, $\vec{b}_3 = (4,14)^T$, $\vec{b}_4 = (12,18)^T$, and $\vec{b}_5 = (14,20)^T$. The grey area is the allowed area for an exact trajectory according to the delta and epsilon values that have been computed from the given base points.

In the example, the six extended base points (considering only dimension j) are $\vec{b}'_0 = (2,0,0,0,2,2)^T$, $\vec{b}'_1 = (6,4,4,4,4,6)^T$, $\vec{b}'_2 = (10,6,0,8,0,11)^T$, $\vec{b}'_3 = (4,0,8,0,8,14)^T$, $\vec{b}'_4 = (12,8,2,8,2,18)^T$, and $\vec{b}'_5 = (14,0,0,10,0,20)^T$. The next figure (Figure 5.3) adds five base points between the existing six points of the previous figure.

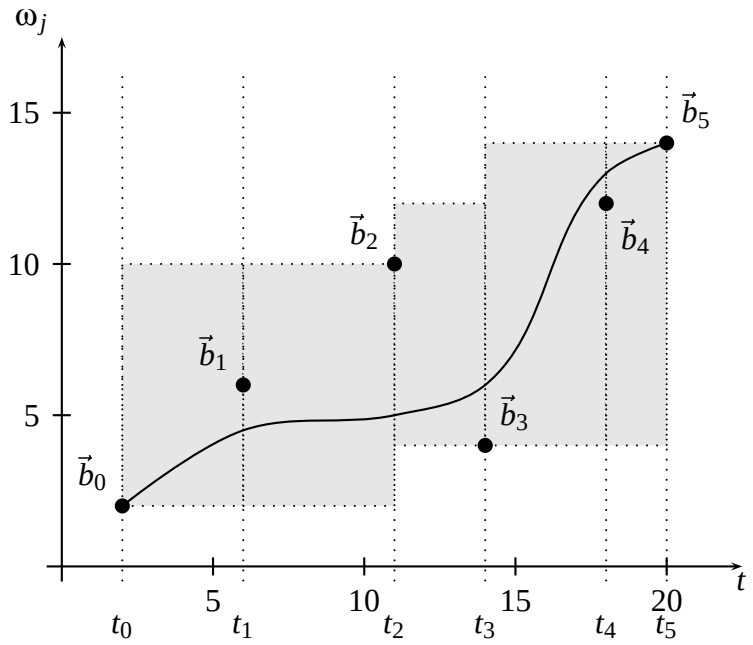

Figure 5.2: *Trajectory defined via six base points.*

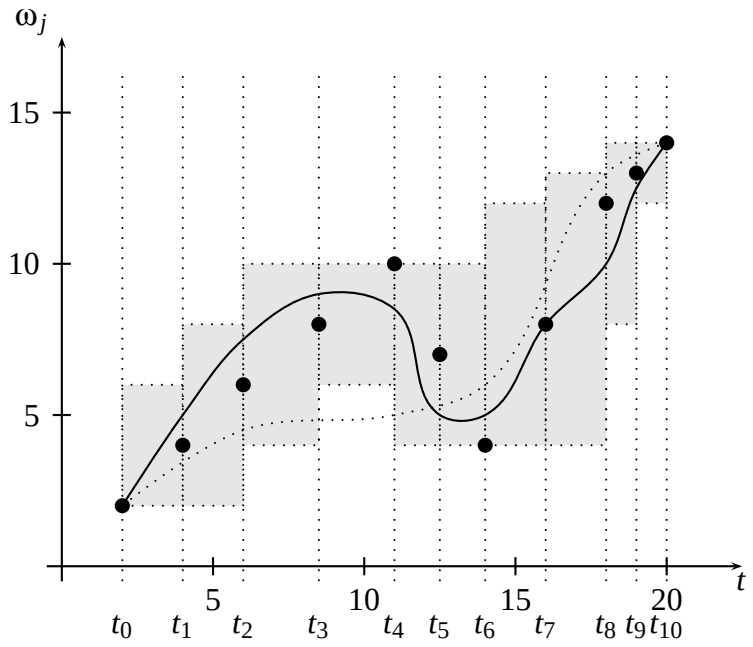

Figure 5.3: *Trajectory defined via eleven base points.*

Here, the allowed region is smaller and the generated trajectory is forced closer to the base points. Note that the old trajectory (dotted line) is now outside the allowed area. Since we always calculate the delta and epsilon values from given base points, we drop them in the extended base point representation and plan using the base points $\vec{b}_i = (\vec{\omega}_i, t_i)^T$ only.

In the next two sections, we present well-known trajectory types that fulfil the epsilon

and delta limits. Firstly, we present the point-to-point and the path motion for a polynomial movement in each joint. Thereafter, we show the same for a modified bang-bang trajectory.

5.3 Polynomial Trajectories

In this section, we focus on the trajectory generation with a polynomial of degree three. In this setting, the movement of each dimension in the configuration space is a polynomial movement. Firstly, the case where the robot stops at each base point is analysed, and afterwards we look at the path motion, where the robot does not stop at every base point. Polynomials with a higher degree could be used as well, but as we only need to get a continuous function for the values and the velocity, a cubic polynomial suffices.

5.3.1 Point-to-Point Motion

Let a base point trajectory be given consisting of $m+1$ base points and let the number of degrees of freedom of the robot be n. If the cubic polynomial (see Figure 5.4) is chosen, then the start configuration $\vec{b}_{i-1}$ and the goal configuration $\vec{b}_i$ of a section and the velocity can be specified. For a point-to-point motion, the velocity in each base point and each dimension is zero. This holds in particular for the start and end configurations. All of the following calculations have to be done for each dimension in the configuration space. Let i be the section number ($i \in \{1,\dots,m\}$) and let j be the dimension ($j \in \{1,\dots,n\}$). Recall that the first section S_0 consists only of the base point $\vec{b}_o$.

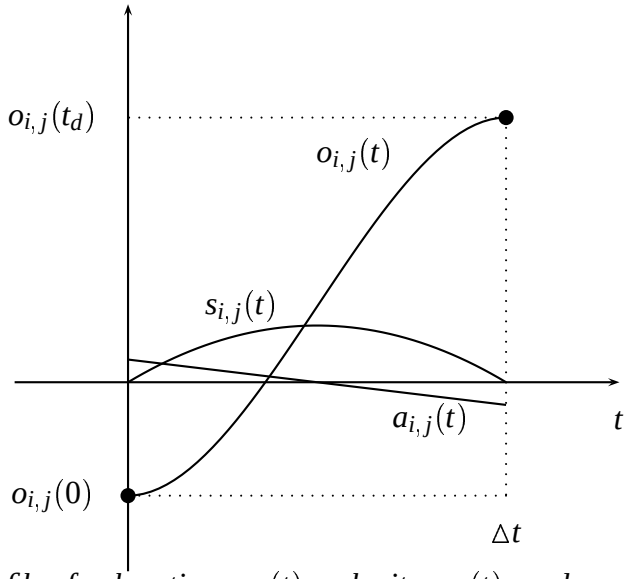

Figure 5.4: *The profiles for location $o_{i,j}(t)$, velocity $s_{i,j}(t)$, and acceleration $a_{i,j}(t)$ of a cubic polynomial.*

For a cubic polynomial the functions of the location $o_{i,j}(t)$, the velocity $s_{i,j}(t)$ and the

acceleration $a_{i,j}(t)$ in the dimension j are

$$\begin{aligned} o_{i,j}(t) &= \breve{a}_{i,j}t^3 + \breve{b}_{i,j}t^2 + \breve{c}_{i,j}t + \breve{d}_{i,j} \\ s_{i,j}(t) &= 3\breve{a}_{i,j}t^2 + 2\breve{b}_{i,j}t + \breve{c}_{i,j} \\ a_{i,j}(t) &= 6\breve{a}_{i,j}t + 2\breve{b}_{i,j} \end{aligned} \tag{5.4}$$

Now the coefficients $\breve{a}_{i,j}$, $\breve{b}_{i,j}$, $\breve{c}_{i,j}$, and $\breve{d}_{i,j}$ of the polynomial have to be calculated. For this calculation, let the starting time be zero ($t_{i-1} = 0$). For the given delta time $\triangle t_i = t_i - t_{i-1}$, the start configuration $\omega_{i-1,j}$, and the goal configuration $\omega_{i,j}$, the values can be put in the equations of (5.4)

$$\begin{aligned} o_{i,j}(0) &= \breve{d}_{i,j} &&= \omega_{i-1,j} \\ s_{i,j}(0) &= \breve{c}_{i,j} &&= 0 \\ o_{i,j}(\triangle t_i) &= \breve{a}_{i,j}\,\triangle t_i^3 + \breve{b}_{i,j}\,\triangle t_i^2 + \omega_{i-1,j} &&= \omega_{i,j} \\ s_{i,j}(\triangle t_i) &= 3\breve{a}_{i,j}\,\triangle t_i^2 + 2\breve{b}_{i,j}\,\triangle t_i &&= 0 \end{aligned} \tag{5.5}$$

Solving the equations results in the following coefficients for the cubic polynomial

$$\begin{aligned} \breve{d}_{i,j} &= \omega_{i-1,j} \\ \breve{c}_{i,j} &= 0 \\ \breve{b}_{i,j} &= 3\frac{\omega_{i,j} - \omega_{i-1,j}}{\triangle t_i^2} \\ \breve{a}_{i,j} &= 2\frac{\omega_{i-1,j} - \omega_{i,j}}{\triangle t_i^3} \end{aligned}$$

We would like to know whether the conditions for delta and epsilon values (5.3) are fulfilled. The values of the exact trajectory are between $\omega_{i-1,j}$ and $\omega_{i,j}$, since we use a cubic polynomial with a first derivation that equals zero at t_{i-1} and t_i.

At time t_{i-1}, the trajectory value is $\omega_{i-1,j}$ and at time t_i, the trajectory value is $\omega_{i,j}$. Hence the polynomial point-to-point motion fulfils the epsilon condition since the epsilon values are always equal to or greater than zero. In the cubic polynomial case, these distances are zero since the exact trajectory is on the base points. Looking at equations (5.3) and (5.2), we note that for the delta values the following holds (because additional base points in the min and max expressions can only increase the delta values):

$$\begin{aligned} \delta_{i,j}^{\min} &\geq \omega_{i,j} - \min\left(\omega_{i-1,j}, \omega_{i,j}\right) \\ \delta_{i,j}^{\max} &\geq \max\left(\omega_{i-1,j}, \omega_{i,j}\right) - \omega_{i,j} \end{aligned}$$

Transforming the inequation and taking into account that the exact trajectory lies between $\omega_{i-1,j}$ and $\omega_{i,j}$, we obtain

$$\begin{aligned} \omega_{i,j} - \delta_{i,j}^{\min} \leq \quad & \min\left(\omega_{i-1,j}, \omega_{i,j}\right) && \leq o_{i,j}(t) \\ o_{i,j}(t) \leq \quad & \max\left(\omega_{i-1,j}, \omega_{i,j}\right) && \leq \omega_{i,j} + \delta_{i,j}^{\max} \end{aligned}$$

From the inequations we can conclude that the polynomial point-to-point trajectory generation fulfils all requirements. The whole trajectory for the dimension j of the configuration space is then defined piecewise as

$$T_j(t) = o_{i,j}(t - t_{i-1}), \quad t_{i-1} < t \leq t_i$$

5.3.2 Path Motion

The disadvantage of the point-to-point motion is that the robot has to come to a full stop at every base point (see Figure 5.5). As a consequence, the maximal necessary acceleration or torque is higher than in a non-stopping movement. Clearly, it seems to be more energy efficient not to stop during a movement but only to reduce speed, unless the direction needs to be changed.

If we allow non-zero velocity in the base points we have to choose the amount of velocity at each base point. Unfortunately, a cubic polynomial is likely to swing (see the second section of the trajectory in Figure 5.6) if the velocity is not zero at the base points. In this example, the velocity at the base point is set to zero if it is a local maximum or minimum regarding the next and previous base point. Otherwise the velocity is the average of the gradients of the previous and next sections of a base point. Here, however, it can happen (as in this example) that parts of the generated trajectory are outside the allowed area.

In the following we only consider one dimension when speaking of the properties of a base point. We suggest calculating the path motion in a different way to avoid the problem of choosing the right velocity to get a non-swinging trajectory run. This is done by first calculating a cubic polynomial between two local maxima and then to stretch or shorten the time. The local maxima/minima are called main base points of dimension j of the trajectory.

Definition 5.1 Main Base Points
A base point $\vec{b}_i = (\vec{\omega}_i, t_i)$ is a main base point of dimension j if

$$\begin{aligned} & i = 0 \\ \vee \quad & i = m \\ \vee \quad & (\omega_{i-1,j} \leq \omega_{i,j} \wedge \omega_{i+1,j} \leq \omega_{i,j}) \\ \vee \quad & (\omega_{i-1,j} \geq \omega_{i,j} \wedge \omega_{i+1,j} \geq \omega_{i,j}) \end{aligned}$$

A main base point $\vec{b}$ of dimension j is responsible for the section starting at the previous main base point for this dimension j and ending at $\vec{b}$.

Definition 5.2 Previous Main Base Point
A base point $\vec{b}_k$ is the previous main base point of a base point $\vec{b}_i$ in the dimension j if $\vec{b}_k, k < i$ is a main base point and

$$\forall \vec{b}_l \text{ with } k < l < i : \vec{b}_l \text{ is no main base point for dimension } j.$$

In the set of "previous base points" of a main base point for dimension j we save the base points which are between the previous main base point and this base point. This set is only defined for main base points of dimension j. For all other base points, this set is empty.

Definition 5.3 Set of Previous Base Points
$\mathcal{P}_{\vec{b}_{i,j}}$ is the set of indices of the base points $\vec{b}_l$ in dimension j belonging to the main base point $\vec{b}_i$. Let $\vec{b}_k$ be the previous main base point of $\vec{b}_i$ in dimension j, then

$$\mathcal{P}_{\vec{b}_{i,j}} = \{l \mid k < l \leq i\}$$

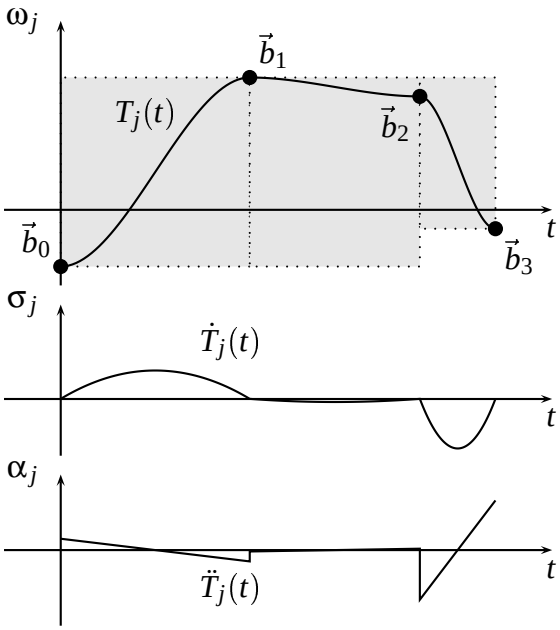

Figure 5.5: *Polynomial point-to-point motion stopping at each base point. The dotted rectangles show the allowed areas for the trajectory*

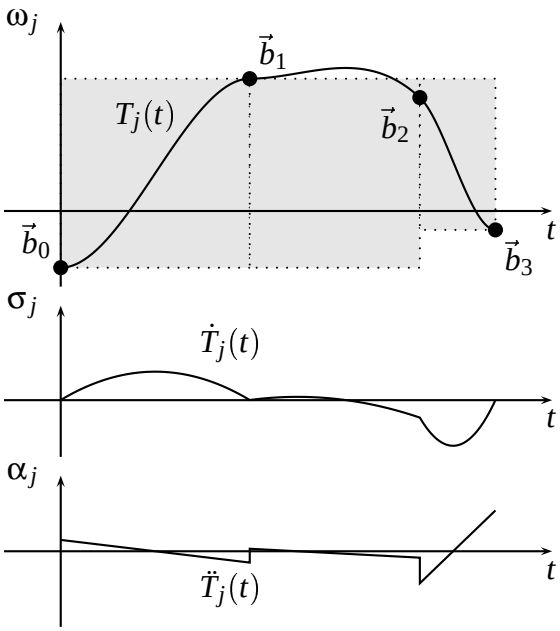

Figure 5.6: *Polynomial path motion when the velocity is an average of the gradients in each base point.*

For example, in Figure 5.6, we have for the given dimension j the main base points $\vec{b}_0$, $\vec{b}_1$, and $\vec{b}_3$. There is no previous main base point for $\vec{b}_0$. The previous main base point of $\vec{b}_1$ is $\vec{b}_0$ and the previous main base point of $\vec{b}_3$ is $\vec{b}_1$. The set of previous base points is defined for $\vec{b}_1$ and $\vec{b}_3$ as $\mathcal{P}_{\vec{b}_{1,j}} = \{1\}$ and $\mathcal{P}_{\vec{b}_{3,j}} = \{2,3\}$ respectively.

For the main base points of each dimension j, we can calculate the functions of the time for the location of a joint $\hat{o}_{i,j}(t)$, the velocity $\hat{s}_{i,j}(t)$ and the acceleration $\hat{a}_{i,j}(t)$ as in the case of a point-to-point motion (the hat-functions only exist for main base points in each dimension and will not be calculated for other base points).

In Figure 5.7, the trajectory is correct. Furthermore, the maximal velocity and acceleration values are smaller or the same as in the point-to-point motion. However it may be that the trajectory is too far away from the base points between two consecutive main base points and does not lie inside the allowed area (see Figure 5.8). In the example, two more base points are added to the trajectory description. Now the new main base points of this dimension are $\vec{b}_0$, $\vec{b}_2$, and $\vec{b}_5$. The sets of the previous base points are now $\mathcal{P}_{\vec{b}_{2,j}} = \{1,2\}$ and $\mathcal{P}_{\vec{b}_{5,j}} = \{3,4,5\}$ (all other sets are empty).

The filled circles are the base points and the hollow circles are the nearest points to the base points lying inside the allowed area at the same point of time. In this invalid case, we have to stretch or shorten the time to make the trajectory be in the allowed area again (see Figure 5.9). This stretching is done by a cubic polynomial.

First, we look for a point $\omega'_{i,j}$ for each value $\omega_{i,j}$ of the base point $\vec{b}_i = (\vec{\omega}_i, t_i)$ at time t_i that is inside the allowed area and that is as close as possible to $\hat{o}_{l,j}(t_i), i \in \mathcal{P}_{\vec{b}_{l,j}}$ (see hollow circles in Figure 5.8). Now the polynomial $\hat{o}_{l,j}(t)$ is changed such that the polynomial touches these points. Let t_i be the time when the value $\omega_{i,j}$ has to be reached. The time $t_{i,j}$ is the time, when the original polynomial $\hat{o}_{l,j}(t)$ reaches the configuration $\omega'_{i,j}$ in dimension j, that is

$$\omega'_{i,j} = \hat{o}_{l,j}(t_{i,j}) \text{ with } i \in \mathcal{P}_{\vec{b}_{l,j}}$$

The next step is to modify the original time run, such that the time point $t_{i,j}$ matches with t_i. Since there are four boundary conditions, a cubic polynomial is needed. Two conditions are the start time and end time and two more conditions have to ensure that the derivations equal one at the beginning and at the end. We also have to make sure that the position and velocity function are still derivable. Hence, each section i needs a function $ct_{i,j}$ for each joint j which changes the time, such that the following conditions are valid. Let $\vec{b}_k$ be the previous main base point of $\vec{b}_i$ in dimension j.

$$\begin{aligned}
ct_{i,j}(t) &= \breve{a}_{i,j}t^3 + \breve{b}_{i,j}t^2 + \breve{c}_{i,j}t + \breve{d}_{i,j} \\
ct_{i,j}(t_{i-1}) &= t_{i-1,j} - t_k \\
\dot{ct}_{i,j}(t_{i-1}) &= 1 \\
ct_i(t_i) &= t_{i,j} - t_k \\
\dot{ct}_{i,j}(t_i) &= 1
\end{aligned}$$

Furthermore we need a monotone accelerating function to ensure that we do not go back in time. In other words, $\dot{ct}_{i,j}(x) \geq 0$. Therefore the changing time function has to

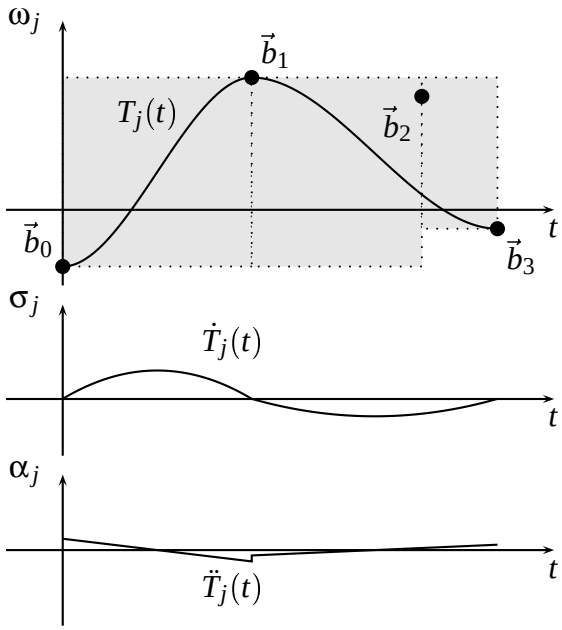

Figure 5.7: *Polynomial path trajectory without consideration of all base points, but only the local maxima and minima. The whole trajectory is inside the allowed area.*

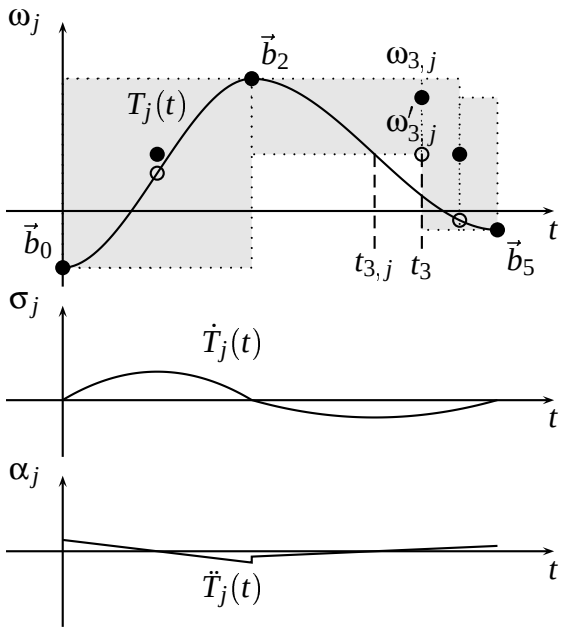

Figure 5.8: *Polynomial path trajectory without consideration of all base points, but only the local maxima and minima. Parts of the trajectory are outside the allowed area.*

fulfil the following condition

$$t_{i,j} - t_{i-1,j} \geq \frac{1}{3}(t_i - t_{i-1}) \tag{5.6}$$

This can be seen when the coefficients $\check{a}, \check{b}, \check{c}$, and $\check{d}$ are calculated and the speed function is analysed, by looking at the derivation and the maximum. For simplicity, take $t_k = 0$, $t_{i-1} = 0$, and $t_{i-1,j} = 0$. Then the first and second derivation of $ct_{i,j}(t)$ are as follows

$$\begin{aligned} \dot{ct}_{i,j}(t) &= 6\frac{t_i - t_{i,j}}{t_i^3}t^2 - 6\frac{t_i - t_{i,j}}{t_i^2}t + 1 \\ \ddot{ct}_{i,j}(t) &= 12\frac{t_i - t_{i,j}}{t_i^3}t - 6\frac{t_i - t_{i,j}}{t_i^2} \end{aligned}$$

Equating the second derivation $\ddot{ct}_{i,j}(t)$ with zero and solving the equations for t results in $t = \frac{1}{2}t_i$. Since it holds that the quadratic polynomial $\dot{ct}_{i,j}$ equals one at the start time and the end time, it is sufficient to examine the first derivation at $t = \frac{1}{2}t_i$.

$$\dot{ct}_{i,j}(\frac{1}{2}t_i) \geq 0 \quad \Rightarrow \quad \frac{t_{i,j}}{t_i} - \frac{1}{3} \geq 0$$

Hence, the maximal compression of the time is one third of the original time. If any of the sections S_l does not fulfil condition (5.6) in dimension j, then $\vec{b}_l$ is taken as a main base point of dimension j as well and the calculation for the configuration dimension j starts again. If all sections fulfil the condition, then the new functions between two base points, with $i \in \mathcal{P}_{\vec{b}_l,j}$, are

$$\begin{aligned} o_{i,j}(t) &= \hat{o}_{l,j}(ct_{i,j}(t)) \\ s_{i,j}(t) &= \hat{s}_{l,j}(ct_{i,j}(t))\dot{ct}_{i,j}(t) \\ a_{i,j}(t) &= 2\hat{a}_{l,j}(ct_{i,j}(t))\dot{ct}_{i,j}(t) + \hat{s}_{l,j}(ct_{i,j}(t))\ddot{ct}_{i,j}(t) \end{aligned}$$

Figure 5.9 shows the modified trajectory with stretched times for the example.

The next figure (Figure 5.10) shows the point-to-point motion for the same example.

Note that in the example, the maximal velocity and acceleration values of the modified trajectory tend to be smaller than the values in a point-to-point motion for a section between two main base points.

Since the first derivation of $ct_{i,j}$ at the intersections to the previous and next section equals one, the speed function is still continuous in base points belonging to the same main base points $i-1, i \in \mathcal{P}_{\vec{b}_l,j}$. Let $\vec{b}_k$ be the previous main base point of $\vec{b}_l$ in dimension j.

$$\begin{aligned} s_{i-1,j}(t_{i-1}) &= \hat{s}_{l,j}(ct_{i-1,j}(t_{i-1}))\dot{ct}_{i-1,j}(t_{i-1}) \\ &= \hat{s}_{l,j}(ct_{i-1,j}(t_{i-1})) \\ &= \hat{s}_{l,j}(t_{i-1,j,n} - t_k) \\ &= \hat{s}_{l,j}(ct_{i,j}(t_{i-1})) \\ &= \hat{s}_{l,j}(ct_{i,j}(t_{i-1}))\dot{ct}_{i,j}(t_{i-1}) = s_{i,j}(t_{i-1}) \end{aligned}$$

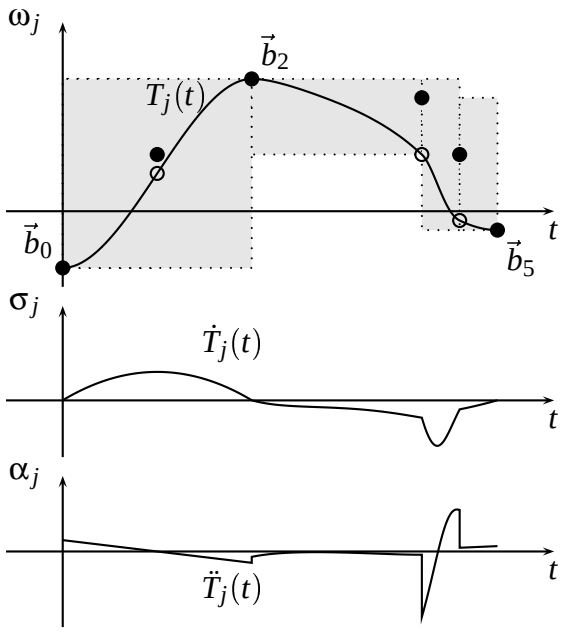

Figure 5.9: *Polynomial path trajectory considering all points. The whole trajectory lies inside the allowed area.*

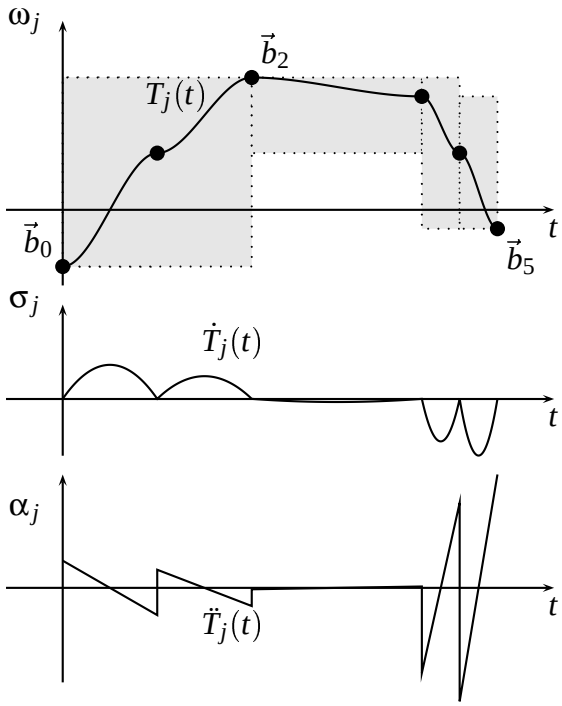

Figure 5.10: *Polynomial point-to-point trajectory considering all points, for the same given base points as in the last figure.*

It is obvious that the resulting trajectory lies inside the allowed area and consequently fulfils the epsilon and delta limits since we have chosen the points $\omega_{i,j}$ inside the allowed area. Second, we made sure that the derivation of the stretching function $ct_{i,j}$ is greater then zero.

5.3.3 Inserting and Deleting Base Points

Since the planning algorithm needs to insert and delete base points, we have to analyse these operations regarding their impact on a polynomial movement. In the following we again look at a single dimension if we speak of a base point $\vec{b}_i$. Hence, the following has to be applied separately to each dimension of the degrees of freedom of the robot.

We first analyse the simple case of a polynomial point-to-point trajectory. If a new base point $\vec{b}_i$ is added in a point-to-point motion, then the old section i has to be replaced by two new sections. The two new sections i and $i+1$ have to be modified. If the i-th base point is deleted, then only the new section i has to be recalculated (see Figure 5.11). Two main points connected via a dotted line represent a variable number of base points but at least one. The solid line connects two base points without any other base point between them and the dashed line represents the recalculation area.

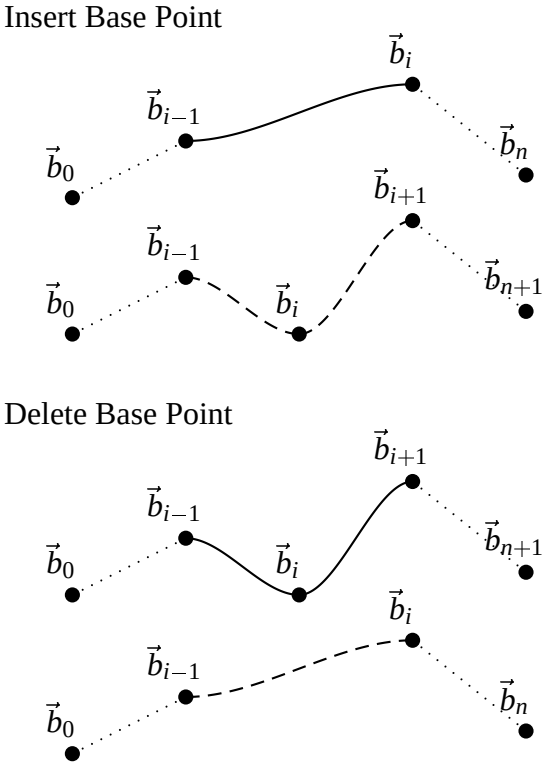

Figure 5.11: *Insertion and deletion of base points in a polynomial point-to-point trajectory.*

If the trajectory movement is a polynomial path motion, then things are more complicated and often more than two sections have to be recalculated, since two main base

points specify the trajectory run. It is obvious that if the old section i of joint j belongs to $\mathcal{P}_{\vec{b}_l,j}$, then at least all of those sections have to be recalculated for joint j. But if the new base point itself is a new main base point, then two or three old main sections might change their values. We now want to take a closer look at the possible situations.

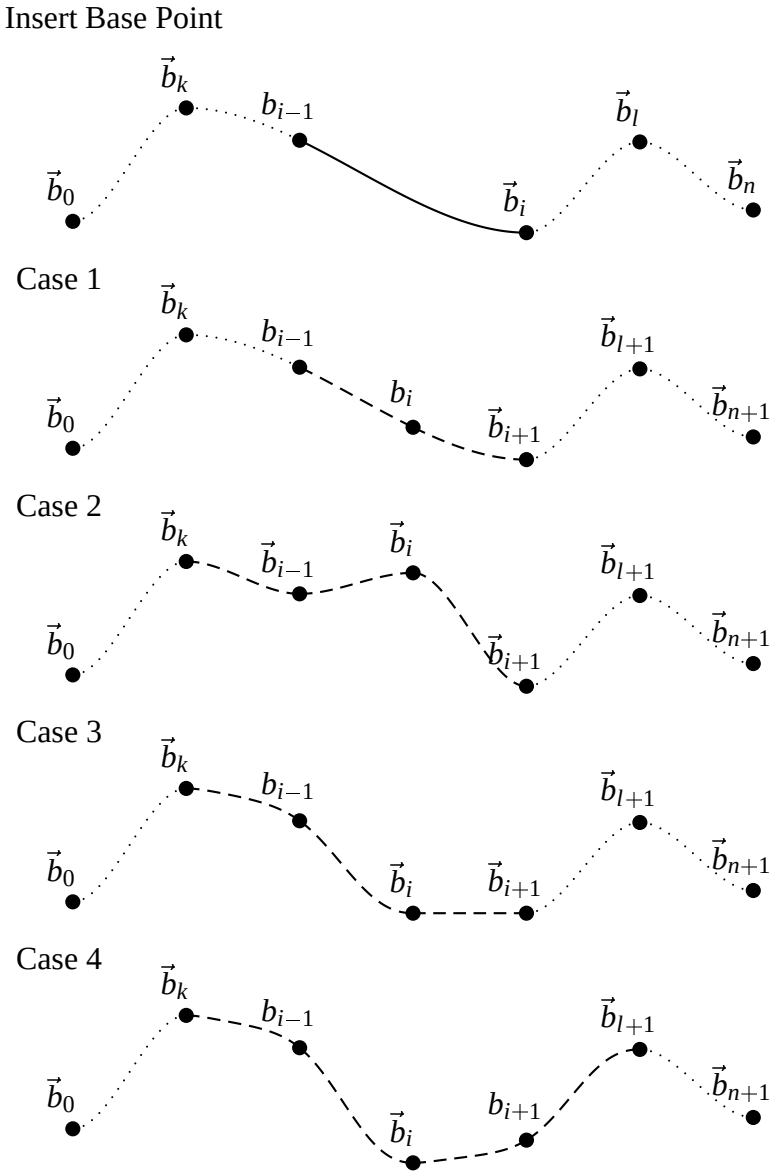

Figure 5.12: *Insertion and deletion of a base point in a polynomial path trajectory.*

In Figure 5.12, we take a look at the case, where $\vec{b}_i$ itself is a main base point and $\vec{b}_{i-1}$ is not a main base point (in dimension j). Furthermore, let $\vec{b}_k$ be the previous main base point of $\vec{b}_i$ and let $\vec{b}_i$ be the previous main base point of $\vec{b}_l$.

In the first case, the new base point $\vec{b}_i$ is not a main base point itself in dimension j, so only the modifying functions of the time for the previous and next section has to be recalculated in this dimension.

If the new base point $\vec{b}_i$ is a main base point, then there are three different cases. The second and third case leaves the other main base points unchanged and therefore only the

sections between $\vec{b}_k$ and $\vec{b}_{i+1}$ change. In the second case, $\vec{b}_{i-1}$ and $\vec{b}_i$ will become a main base point as well. In the third case only $\vec{b}_i$ becomes a main base point. Now not only the modifying functions of the time have to be recalculated but the hat-functions of the new main base points and $\vec{b}_{i+1}$ need recalculation too.

The fourth case changes $\vec{b}_i$ to a normal base point and the trajectory run changes from $\vec{b}_k$ to $\vec{b}_{l+1}$.

If $\vec{b}_{i-1}$ is a main base point and $\vec{b}_i$ is not a main base point, then we have the mirrored cases. If neither $\vec{b}_{i-1}$ nor $\vec{b}_i$ are main base points, then only case one, case two, and the mirrored case two would occur. If both $\vec{b}_{i-1}$ and $\vec{b}_i$ are main base points, then it could happen that three main sections change their values, if you think of inserting a point in section $i+1$ in case three that is lower than points $\vec{b}_i$ and $\vec{b}_{i+1}$.

If a base point gets deleted we again have to distinguish whether a base point or a main base point is deleted. In the first case, a normal base point leads to a recalculation of modifying functions of the time of the previous and next section (reverting case one of Figure 5.12). In the second case, the deletion of a main base point, it is more complicated. Just think of reverting the insertion in Figure 5.12.

To summarise, we can say that the path motion may lead to a trajectory run with less velocity and acceleration, but the adding and deleting is quite costly. Especially as the number of affected sections is not constant (as in the point-to-point motion) and considering the fact that the above operations have to be performed for each dimension separately. Nevertheless, the method presented here ensures a non-swinging trajectory, hence the motion stays in the allowed area defined by the given base points.

5.4 Modified Bang-Bang Trajectories

In a bang-bang trajectory, the third derivation of the trajectory of one configuration dimension is always zero. This means that the acceleration is piecewise constant and jumps from one value to another. In a traditional bang-bang trajectory, the acceleration value is either zero, minimal, or maximal. In our case, the acceleration can be of any value. In the next two subsections, we analyse the point-to-point motion and the path motion.

5.4.1 Point-to-Point Motion

If the path motion stops at every control point of the path (see Fig. 5.13), we have a trapezoidal velocity profile. A point-to-point motion is split into an acceleration phase $[t_{i-1}, t_{i-1}+t_a]$, a constant speed phase $[t_{i-1}+t_a, t_i-t_a]$, and a negative acceleration phase $[t_i-t_a, t_i]$. We assume that the acceleration duration t_a is given. For example, t_a could be chosen half the distance between two consecutive base points ($t_a = (t_{i-1}+t_i)/2$) or smaller.

$$o_{i,j}(t) = \begin{cases} \omega_{i-1,j} + \frac{1}{2}\check{a}_{i,j}(t-t_{i-1})^2 & : \quad \text{if } t_{i-1} \le t \le t_{i-1}+t_a \\ o_{i,j}(t_{i-1}+t_a) + \check{s}_{i,j}(t-(t_{i-1}+t_a)) & : \quad \text{if } t_{i-1}+t_a < t \le t_i - t_a \\ \omega_{i,j} - \frac{1}{2}\check{a}_{i,j}(t_i-t)^2 & : \quad \text{if } t_i - t_a \le t \le t_i \end{cases} \tag{5.7}$$

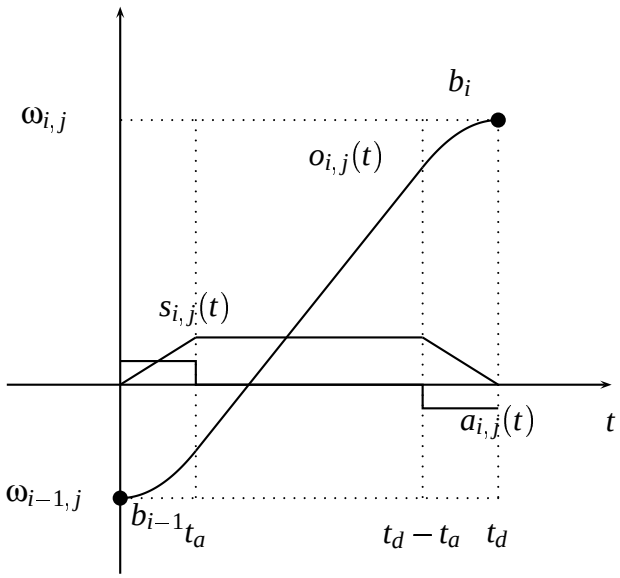

Figure 5.13: *The profiles for value, speed, and acceleration of a bang-bang trajectory.*

We now have to calculate $\check{a}_{i,j}$ and $\check{s}_{i,j}$. $\check{s}_{i,j}$ is the speed at the end of the acceleration phase, which is equal to the first derivation of $o_{i,j}(t)$ at time $t_{i-1}+t_a$.

$$\check{s}_{i,j} = \dot{o}_{i,j}(t_{i-1}+t_a) = \check{a}_{i,j}t_a$$

From the following equation we can calculate $\check{a}_{i,j}$. On the left side the difference of the values are taken and on the right side the sum of the absolute motion of the three motion functions (5.7) is built. Let $\triangle t_i$ be the delta distance between the two base points ($\triangle t_i = t_i - t_{i-1}$).

$$\begin{aligned}
\omega_{i,j} - \omega_{i-1,j} &= \frac{1}{2}\check{a}_{i,j}t_a^2 + \check{s}_{i,j}(t_i - t_a - t_{i-1} - t_a) + \frac{1}{2}\check{a}_{i,j}t_a^2 \\
&= \check{a}_{i,j}t_a^2 + \check{a}_{i,j}t_a\ \triangle t_i - 2\check{a}_{i,j}t_a^2 \\
&= \check{a}_{i,j}t_a(\triangle t_i - t_a) \\
\check{a}_{i,j} &= \frac{\omega_{i,j} - \omega_{i-1,j}}{t_a(\triangle t_i - t_a)}
\end{aligned}$$

The velocity and acceleration functions follow from the derivations of the function $o_{i,j}(t)$.

$$s_{i,j}(t) = \begin{cases} \check{a}_{i,j}(t - t_{i-1}) & : \text{ if } t_{i-1} \le t \le t_{i-1}+t_a \\ \check{s}_{i,j} = \check{a}_{i,j}t_a & : \text{ if } t_{i-1}+t_a < t < t_i - t_a \\ -\check{a}_{i,j}(t_i - t) & : \text{ if } t_i - t_a \le t \le t_i \end{cases} \tag{5.8}$$

$$a_{i,j}(t) = \begin{cases} \check{a}_{i,j} & : \text{ if } t_{i-1} \le t \le t_{i-1}+t_a \\ 0 & : \text{ if } t_{i-1}+t_a < t < t_i - t_a \\ -\check{a}_{i,j} & : \text{ if } t_i - t_a \le t \le t_i \end{cases} \tag{5.9}$$

Since the trajectory of a section lies between the previous and the next base point, we can use the same arguments as for the point-to-point motion with a polynomial (Section 5.3.1) to show that the requirements are fulfilled.

5.4.2 Path Motion

If the path consists of only two base points, then the trajectory run is like a point-to-point motion, except that we take $t_a = \frac{t_1 - t_0}{2}$. Otherwise we do not want to stop at every point. Therefore, we get from one to the next section by a cubic polynomial (see Figure 5.14).

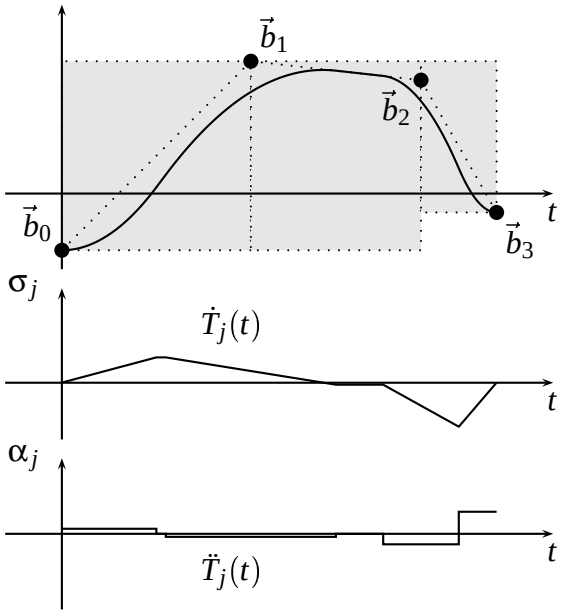

Figure 5.14: *The path motion without stopping at each base point for a bang-bang trajectory.*

We need at least a cubic polynomial since we have to satisfy four conditions. These are the configurations when the blending starts and ends and the velocities at these points. Let $g_{i,j}$ be the gradient of section i in dimension j and let $t_{a,i}$ be the maximal delta time a joint may leave the direct connection. This means that the blending starts at $t_i - t_{a,i}$ and finishes at $t_i + t_{a,i}$.

$$g_{i,j} = \begin{cases} 0 & : \text{ if } i = 0, m+1 \\ \frac{2}{3}\frac{\omega_{i,j} - \omega_{i-1,j}}{t_i - t_{i-1}} & : \text{ if } i = 1, m \\ \frac{\omega_{i,j} - \omega_{i-1,j}}{t_i - t_{i-1}} & : \text{ if } 1 < i < m \end{cases} \tag{5.10}$$

$$t_{a,i} = \begin{cases} \frac{t_1 - t_0}{2} & : \text{ if } i = 0 \\ \frac{t_m - t_{m-1}}{2} & : \text{ if } i = m \\ min\{\frac{t_i - t_{i-1}}{2}, \frac{t_{i+1} - t_i}{2}\} & : \text{ if } 0 < i < m \end{cases} \tag{5.11}$$

As there is no section before the first and after the last section, we have to treat them separately. We can now calculate the coefficients of the cubic polynomial $f_{i,j}(t)$ for the duration from $[t_i - t_{a,i}, t_i + t_{a,i}]$ with the following equations:

$$\begin{aligned} f_{i,j}(t) &= \breve{a}_{i,j}t^3 + \breve{b}_{i,j}t^2 + \breve{c}_{i,j}t + \breve{d}_{i,j} \\ f_{i,j}(t_i - t_{a,i}) &= \omega_{i,j} - g_{i,j}t_{a,i} \\ f_{i,j}(t_i + t_{a,i}) &= \omega_{i,j} + g_{i+1,j}t_{a,i} \\ \dot{f}_{i,j}(t_i - t_{a,i}) &= g_{i,j} \\ \dot{f}_{i,j}(t_i + t_{a,i}) &= g_{i+1,j} \end{aligned} \tag{5.12}$$

Solving these equations results in $\breve{a}_{i,j} = 0$, so the polynomial is quadratic. Therefore, the first derivation is a straight line for speed and the second derivation is a constant value for acceleration. This means that the trajectory is still a bang-bang trajectory. The blending functions are defined for $0 < i < m$. For the first and the last base point the blending functions are different since we do not have a previous or next section and hence we have to touch these base points. For $i = 0$ and $i = m$, the following equations need to be solved:

$$\begin{aligned} f_{0,j}(t_0) &= \omega_{0,j} \\ f_{0,j}(t_0 + t_{a,0}) &= \frac{2\omega_{0,j} + \omega_{1,j}}{3} \\ \dot{f}_{0,j}(t_0) &= 0 \\ \dot{f}_{0,j}(t_0 + t_{a,0}) &= g_{1,j} \\ f_{m,j}(t_m - t_{a,m}) &= \frac{2\omega_{m,j} + \omega_{m-1,j}}{3} \\ f_{m,j}(t_m) &= \omega_{m,j} \\ \dot{f}_{m,j}(t_m - t_{a,m}) &= g_{m,j} \\ \dot{f}_{m,j}(t_m) &= 0 \end{aligned}$$

Again, solving the equations results in $\breve{a}_{i,j} = 0$ and therefore a constant acceleration. The section $]t_{i-1} + t_{a,i-1}, t_i - t_{a,i}[$ between the parabolic blend movement is a linear movement.

In Figure 5.15 and Figure 5.14 the difference between stopping at the base points and not stopping can be seen. The absolute accelerations are getting smaller for a path motion. That the requirements are still fulfilled is obvious, as the linear connections between two base points are tangents to the blend functions.

5.4.3 Inserting and Deleting Base Point

If the trajectory movement is a point-to-point motion and a new base point is inserted in section *i*, then only this section has to be replaced by two new sections. Deleting a base point *i* leads to recalculation of one new section. This is the same situation as in the polynomial point-to-point motion (Figure 5.11, page 58).

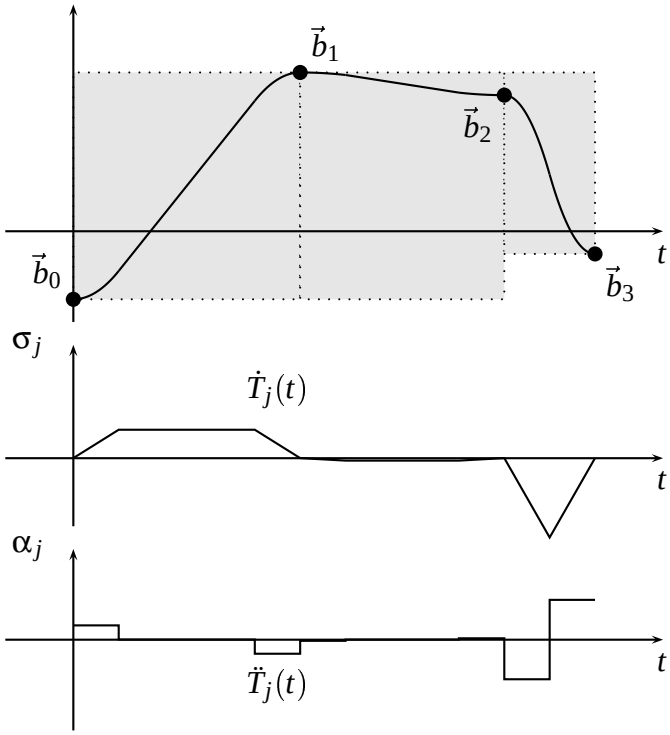

Figure 5.15: *The corresponding point-to-point motion stopping at each base point.*

On the other hand, if it is a path motion, then the section i has to be replaced by two new sections and the previous $i-1$ and next sections $i+1$ have to be recalculated as well (Figure 5.16). As before, the dashed line indicates the region that needs to be recalculated.

This is because the blending function needs two consecutive sections to be calculated. Deleting a base point leads to a new section i. So the previous and next section of this new section change the run of the trajectory as well. This is much better than in the polynomial path motion. In the polynomial path motion up to four sections need to be modified when adding a base point.

5.5 Summary and Outlook

In this chapter, the definition of a base point trajectory has been introduced, which describes allowed areas for an exact trajectory in the configuration space. The advantage of this approach is that our planning process is decoupled from the actual trajectory type. With each new base point the planning process gets more control on the actual run of the exact trajectory. We have shown using two well-known types of trajectories, how an exact trajectory can be generated such that the trajectory run is inside the allowed area. We have analysed the operation of deletion and insertion of base points into an existing base point trajectory. Here our focus was on the locality of the changes that are necessary to update an existing exact trajectory after these operations. Our conclusion is that recalculations after changing base points in the polynomial point-to-point movement and in the modi-

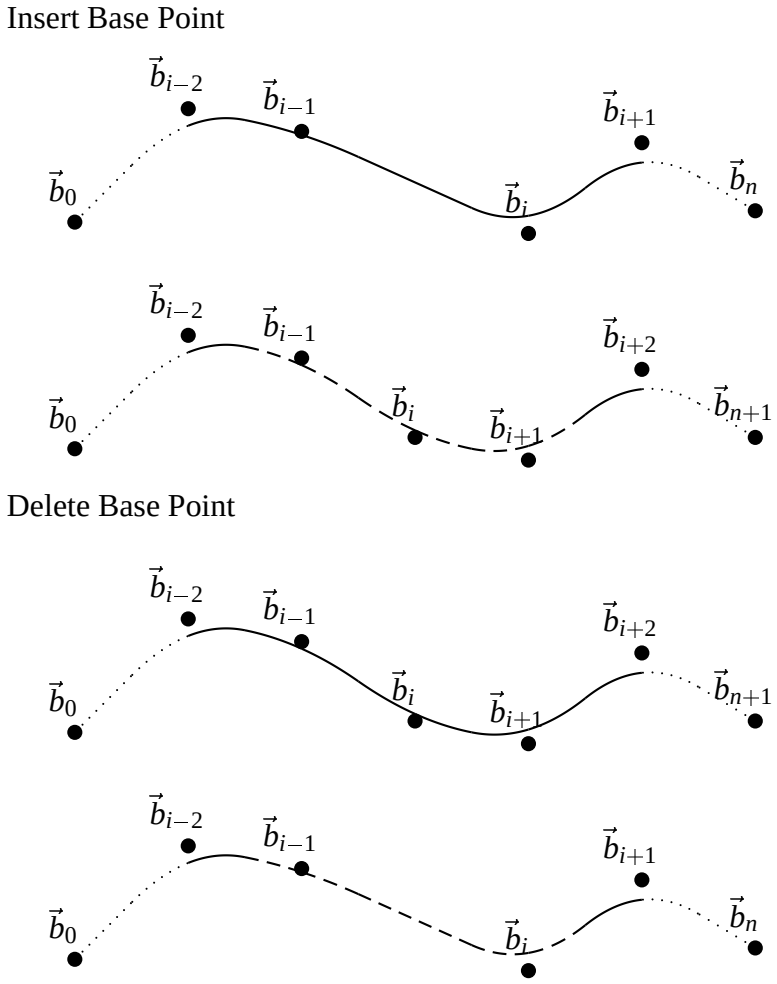

Figure 5.16: *Insertion and deletion of a base point in a bang-bang path trajectory.*

fied bang-bang motions are much easier than in a polynomial path motion. The modified bang-bang path motion seems to be the best compromise between smoothness of motion and the locality of necessary modifications.

Let us now look at other trajectory types. Another promising candidate for robot motion are B-Splines. Their behaviour is very similar to our modified bang-bang path motion. It should be possible to use this motion type in our trajectory generation, although some consideration might be necessary regarding how the B-splines are defined in the start and goal configurations.

A more interesting example are trajectories for mobile robots. In the following, we give hints on how our base point trajectory approach may be used for planning motions of mobile robots. Let the mobility of a car-like robot be modelled with three joints as in Figure 4.6 on page 36. Then a trajectory might be generated (in physical space) by using the orientation of the robot as tangents of a cubic polynomial (Figure 5.17). The circles indicate the moving direction of the robot. The mobile robot tries to move from the start configuration (depicted as robot with filled circle) to the goal configuration (robot with hollow circle). For the left and the middle image, such a motion with a cubic polynomial

can be given.

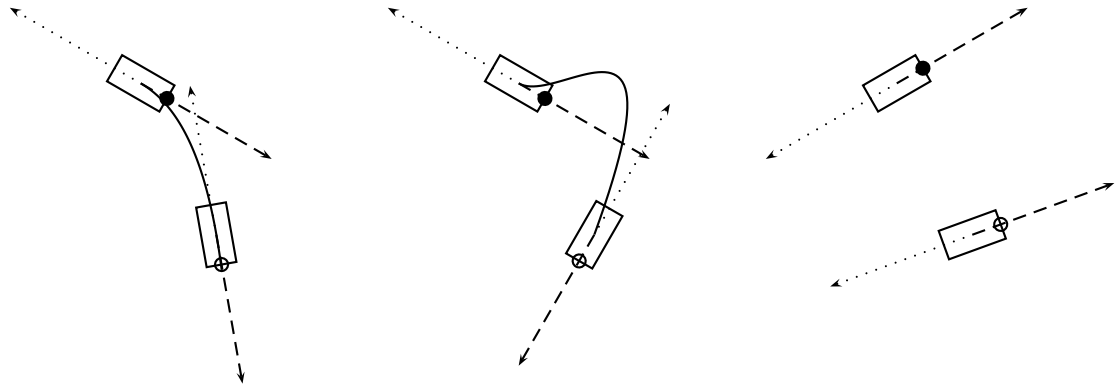

Figure 5.17: *Mobile robots' trajectories.*

In [Buttelmann et al., 1999], trajectory generation for non-holonomic autonomous car-like robots with straight lines, circles, and cubic polynomials is analysed.

Let us now consider a base point trajectory that defines allowed areas for the exact trajectory, as in Figure 5.18. If a motion between base points leaves the allowed area (middle) or cannot be given (right), then we suggest to take the edges of the allowed area as tangents to get an exact trajectory in the allowed area. Hence, for the robot in the start configuration (robot with black circle) we consider the arrow in the moving direction and the two edges of the allowed area that the robot touches. If the arrow in the moving direction points outside the allowed area, then we drop it, otherwise the edge is dropped with the smaller angle between the edge and the arrow. For the goal configuration (robot with hollow circle) the same is done, but the arrow is opposite to the moving direction. Now two possible trajectories for each situation can be generated (solid and dashed curves), by taking the remaining lines as tangents for the curves.

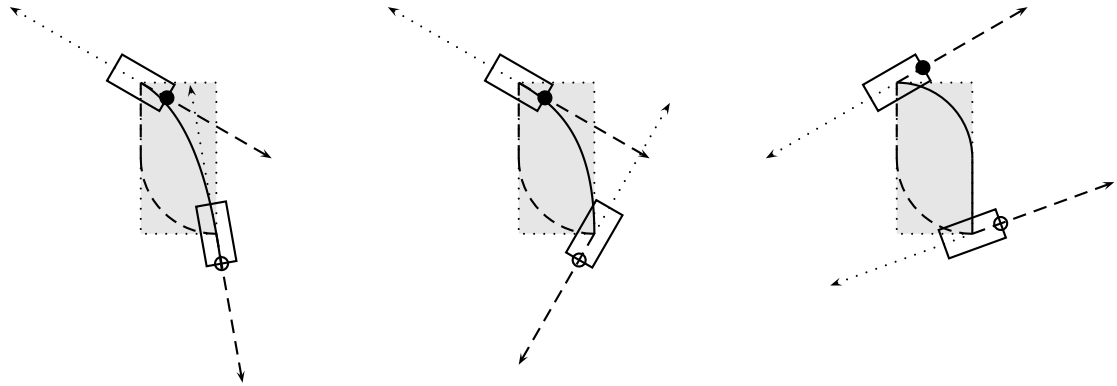

Figure 5.18: *Mobile robots' trajectories inside the allowed areas.*

The idea is, that although these trajectories may not be traceable, we can use them in our motion planner. We do not say that these trajectories cannot be tracked because of kinematic constraints, but because of dynamic constraints. In other words, we pretend

that a motion against the wheels' direction exceeds the robot's force or torque limits, and we let our rating function discard those trajectories that cannot be tracked by giving them a bad rating. To use our technique, we need a rating function that rates the "severity" of a movement against the wheels. We conjecture that such a rating would not be difficult to design.

In the above thoughts, we did not consider the violation of the interval for the orientation angle of the mobile robot. This means either the robot has to move clockwise or in opposite direction viewed from the top. This can be solved by choosing the right curve in the allowed area in Figure 5.18. The solid lines are used when the angle of the following orientation is greater than the angle of the current orientation (clockwise) and the dashed lines are used if the angle of the next orientation is smaller than the old one (counter-clockwise).

This is certainly not the best possible procedure for planning motions of mobile robots, but it shows the generality of our planning approach.

CHAPTER 6

Force and Torque Rating

In this section, we present a trajectory rating that reflects violation of robot dynamics. We assume that gravity is present and that the maximal forces and torques of the robot's joints are known. We first explain the Newton-Euler algorithm which is used to determine the torque. We then discuss different aspects of a good trajectory.

6.1 Introduction

The base point trajectory definition guarantees that in the exact trajectory the configuration space limits (and hence the joint limits) are kept. What remains is to rate the quality of the trajectory regarding the dynamic limits of the robot. To this end, we need a dynamic model of the robot.

An algorithm for modelling robot dynamics that is well-suited for forked manipulator chains has been developed by Murray and Neuman [Murray and Neumann, 1986]. We give a short review of their approach in the following section. The next step is to define a rating that reflects the actual force and torque and the given dynamic limits of the robot. We give such a rating function in the section thereafter.

6.2 Force and Torque Calculation

For the different types of trajectories we have to calculate the torque in each joint of the robot. The Newton-Euler algorithm of Murray and Neuman [Murray and Neumann, 1986] can calculate the actual torque of each joint in linear time, doing a forward and backward calculation along the robot's topology. Special joints or mobile robots are not considered.

The Newton-Euler algorithm needs the mass m_i and the inertia tensor I_i of each link L_i with respect to its centre of gravity g_i (given in the coordinate system of joint J_i and L_i respectively). In [Mirtich, 1996] an algorithm for computing polyhedral mass properties can be found. This algorithm calculates the mass, the centre of gravity and the inertia

tensor, if the density of a body is known and the shape of the body is given via polyhedrals. This is much easier than to give an inertia tensor. The resulting dynamic model is only an approximation of the dynamic model of a real robot, which may have an inhomogeneous mass distribution. Nevertheless, the approximation is sufficient for our purposes.

The forward recursion of the Newton-Euler algorithm has to calculate the angular velocity $\breve{\sigma}_j$, the angular acceleration $\breve{\alpha}_j$ and linear acceleration $\breve{\alpha}_j$ relative to the world for each joint J_j of the robot. We will give a short overview of the algorithm, for a detailed description we refer to [Pu, 1998].

We assume that the gravity g is in negative z direction of the world coordinate system. The function $p(j)$ returns the index of the previous joint of J_j. Let for each joint the current position $\upsilon_j = \upsilon_{\omega,j}$, the current speed $\sigma_j = \upsilon_{\sigma,j}$, and the current acceleration $\alpha_j = \upsilon_{\alpha,j}$ be given. The type of joint J_j is determined by λ_j, with $\lambda_j = 1$ if it is a revolution joint and $\lambda_j = 0$ if it is a prismatic joint. $R_j(\upsilon_j)$ is the rotation 3×3 matrix converting vectors from the coordinate system of the previous joint in the coordinate system of the current joint. Hence, this is the upper left 3×3 matrix of $({}^{p(j)}D_j \; {}^{j}E(\upsilon_j))^{-1}$ (see Section 4.1.1, page 30). The rotation matrix is constant for prismatic joints. The translation vector $p_j(\upsilon_j)$ is the vector directed from the coordinate system from $J_{p(j)}$ to J_j. This is the translational part of $({}^{p(j)}D_j \; {}^{j}E(\upsilon_j))$. This translation vector is constant for revolute joints.

The initial values for the forward recursive equations are

$$\begin{aligned} \vec{z} &= (0,0,1)^T \\ \vec{\breve{\sigma}}_0 &= (0,0,0)^T \\ \vec{\breve{\alpha}}_0 &= (0,0,0)^T \\ \vec{\breve{\alpha}}_0 &= (0,0,-g)^T \end{aligned}$$

This means that there is no angular velocity or acceleration, but a linear acceleration because of gravitation. The forward recursive equations for $j = 1,\ldots,n$ are

$$\vec{\breve{\sigma}}_j = R_j(\upsilon_j)\vec{\breve{\sigma}}_{p(j)} + \lambda_j \vec{z}\sigma_j \tag{6.1}$$

$$\vec{\breve{\alpha}}_j = R_j(\upsilon_j)\vec{\breve{\alpha}}_{p(j)} + \lambda_j \left(\left(R_j(\upsilon_j)\vec{\breve{\sigma}}_{p(j)} \right) \times \left(\vec{z}\sigma_j \right) + \vec{z}\alpha_j \right) \tag{6.2}$$

$$\begin{aligned} \vec{\breve{\alpha}}_j = {} & R_j(\upsilon_j) \left(\vec{\breve{\alpha}}_{p(j)} + \vec{\breve{\alpha}}_j \times p_j(\upsilon_j) + \vec{\breve{\sigma}}_j \times \left(\vec{\breve{\sigma}}_j \times p_j(\upsilon_j) \right) \right) \\ & + \left(1 - \lambda_j\right) \left(\vec{z}\alpha_j + 2 \left(R_j(\upsilon_j)\vec{\breve{\sigma}}_{p(j)} \right) \times \left(\vec{z}\sigma_j \right) \right) \end{aligned} \tag{6.3}$$

In Equation (6.1) the angular velocity of joint J_j is derived from the angular velocity of the previous joint and the angular velocity of the current joint if it is a revolute joint. The angular acceleration (6.2) combines the angular acceleration of the previous joint and, if the current joint is a revolution joint, the acceleration of the joint and the acceleration rate resulting from the combination of the previous speed and the speed of the current axis.

The linear velocity (6.3) is derived from the previous linear acceleration and the parts from the angular velocity and acceleration. If the current joint is a prismatic joint, then we have to add the speed and acceleration parts of the current joint.

The local equations for each link $j = 1,\ldots,n$ calculate the translational force F_j and the rotational torque N_j in the centre of gravity of each link, without considering other links.

$$F_j = m_j\left(\vec{\alpha}_j + \vec{\dot{\alpha}}_j \times g_j + \vec{\dot{\sigma}}_j \times \left(\vec{\dot{\sigma}}_j \times g_j\right)\right) \tag{6.4}$$

$$N_j = I_j\vec{\dot{\alpha}}_j + \vec{\dot{\sigma}}_j \times \left(I_j\vec{\dot{\sigma}}_j\right) \tag{6.5}$$

The force calculation (6.4) multiplies the mass with the acceleration amount in the centre of gravity. The torque (6.5) needs the inertia tensor of the link L_j, and the angular velocity and acceleration.

The backward recursive equations for $j = n,\ldots,1$ calculate the translational force f_j and the rotational torque n_j in each joint. τ_j stores the force and the torque respectively of each joint.

$$f_j = F_j + \sum_{\forall i:p(i)=j} R_i^{-1}(\upsilon_i)\, f_i \tag{6.6}$$

$$n_j = N_j + g_j \times F_j + \sum_{\forall i:p(i)=j} \left(R_i^{-1}(\upsilon_i) n_i + p_i(\upsilon_i) \times \left(R_i^{-1}(\upsilon_i)\, f_i\right)\right) \tag{6.7}$$

$$\tau_j = \lambda_j n_j^T \vec{z} + \left(1-\lambda_j\right) f_j^T \vec{z} \tag{6.8}$$

The force (6.6) in joint J_j needs to be the sum of all following forces and its own force. This is almost the same for the torque (6.7) in joint J_j, but considers the torques as well. Finally, (6.8) gives the force and the torque respectively in joint J_j. This completes the torque and force calculation at discrete time steps, provided that the positions, velocities and accelerations in each joint are given.

6.3 Trajectory Force and Torque Rating

Let $M : \mathbb{R}^{3n} \to \mathbb{R}^n$ be the function which represents the dynamic model of the robot and let $\vec{\upsilon}_d$ be a dynamic configuration. Then $M(\vec{\upsilon}_d)$ returns the forces and torques $\vec{\tau}$ in the joints of the robot.

Since we are interested in the joints violating the maximal force or torque limit $\theta_j^{\max}$ during the trajectory run, all these joints are considered in the rating. Let $\delta : \mathbb{R} \to \mathbb{R}$ be a function which clips the negative values

$$\delta(x) = \begin{cases} 0 & : \quad \text{if } x \leq 0 \\ x & : \quad \text{if } x > 0 \end{cases}$$

The rating of a dynamic configuration $\vec{\upsilon}_d$ is then given by

$$r_t^s = \sum_{j=1}^{n} \frac{\delta(\mid \tau_j \mid -\theta_j^{\max})}{\theta_j^{\max}}, \text{ with } \vec{\tau} = M(\vec{\upsilon}_d) \tag{6.9}$$

This rating has two important properties. First, the amount of capacity overload is set in relation to the maximum capacity of the joint. Second, taking the sum gives the possibility

of a subtle graduation in the rating, because even an improvement in a single joint is detected. This has some advantages compared to taking the maximum of all overloads, where the worst joint has to improve the situation. If the rating is zero, then the dynamic configuration is valid and therefore traceable.

Finally, we are interested in the rating of a whole trajectory $T(t)$ between a start time t_s and a goal time t_g. Let $T^D(t)$ be the dynamic configuration $\vec{\upsilon}_d$ at time t of the trajectory $T(t)$ ($\vec{\upsilon}_d = T^D(t)$, see Chapter 4.4). The rating of a trajectory section T_i is then given by the maximum of all ratings

$$r_t = \max\left(\sum_{j=1}^{n} \frac{\delta(\mid M(T^D(t))_j \mid -\theta_j^{\max})}{\theta_j^{\max}} : \text{ with } t_{i-1} \leq t \leq t_i\right) \tag{6.10}$$

Since the calculation is only possible at a discrete point in time, we have to choose these points carefully. Testing at equal distributed time points results in a inaccurate approximation of the real value if the number of tests is small. This is because fast movements would be tested at fewer points in time than slow movements, since the robot moves larger distances if it moves fast. Quite the opposite should be the case. We should test more often during fast movements, because the changes in the forces and torques are higher during a fast movement, for example, think of the influence of the gravity on the forces in joints during movement.

We suggest a test that depends only on the link movements. We test compliance of dynamic limits at time configurations such that the maximal distance of any point in the robot's joints does not exceed a distance p between two consecutive tested time configurations. In this case, we say the calculation of the rating is done with precision p.

For each link L_l we choose three reference points x_l, y_l, and z_l relative to the origin of J_l, which approximate the size and the position of the link. In other words, the triangle given by the three points is a placeholder for the link. One point is in the origin $x_l = (0,0,0)^T$, the second gives the expansion

$$y_l = (\max(|v_{t,1}|), \max(|v_{t,2}|), \max(|v_{t,3}|))^T$$

with $t \in L_l$, and the third is for keeping the orientation $z_l = (-y_{l,1}, -y_{l,2}, y_{l,3})^T$ (All values should be unequal zero if the link has a distribution). The idea is to analyse the motion of these points during a time interval for each link in the real world. If we look at the configuration at t_s and t_g, no reference point of any link is allowed to move further then p, otherwise the time interval is bisected.

Let *double* `getMaxMovement` (*trajectory T*, *double* t_s, *double* t_g) be the function that returns the maximal distance between two consecutive reference points if the robot is in position $T(t_s)$ and $T(t_g)$. If ${}^0x_l^s, {}^0y_l^s$, and ${}^0z_l^s$, with $l = 1,\ldots,n$ are the positions of the reference points relative to the world when the robot is at time t_s and ${}^0x_l^g, {}^0y_l^g$, and ${}^0z_l^g$ are their positions at goal time t_g, then `getMaxMovement` returns

$$\max\left(|{}^0x_l^g - {}^0x_l^s|, |{}^0y_l^g - {}^0y_l^s|, |{}^0z_l^g - {}^0z_l^s| : l = 1,\ldots,n\right).$$

We would like to note that the positions of the reference points belonging to link L_l depend on the set of previous joints $\mathcal{L}_l^{\mathbf{pj}}$ of link L_l. Consequently, for each call of

`getMaxMovement` the values of the exact trajectory and from them the positions of the reference points have to be calculated at time t_s and t_g.

The function *double* `rateTorqueAndForce` (*trajectory* T, *double* t_s, *double* t_g) returns the rating r_t of the trajectory section lying between t_s and t_g for a global given precision p according to Equation 6.10. The calculation is done recursively by an embedded function.

double `rateTorqueAndForce` (*trajectory* T, *double* t_s, *double* t_g)
pre-condition: $t_s < t_g$

```
/* calculate force and torque at time t_s (Newton-Euler algorithm) */
double[] τ = M(T^D(t_s))

/* calculate force and torque rating r_s according to Equation 6.9 */
for (int i = 1, i ≤ n, i++)
    if (| τ_i |> θ_i^max)
        τ_i = (| τ_i | −θ_i^max)/θ_i^max
    else
        τ_i = 0

/* calculate force and torque between ]t_s,t_g] */
rateTorqueAndForceEmbedded(T,t_s,t_g,τ)

/* sum maximum forces and torques of all joints */
double r = 0
for (int i = 1, i ≤ n, i++)
    r+ = τ_i
return r
```

The above function `rateTorqueAndForce` first calculates the force and torque at time t_s and uses the embedded function *void* `rateTorqueAndForceEmbedded` (*trajectory* T, *double* t_s, *double* t_g, *double*[] τ) to calculate the torque and force rating for the rest of trajectory T recursively. τ stores the maximal forces and torques in each joint.

void `rateTorqueAndForceEmbedded` (*trajectory* T, *double* t_s, *double* t_g, *double*[] τ)
pre-condition: $t_s < t_g$

```
/* check if the maximal movement of each reference point is too large */
if (getMaxMovement(T,t_s,t_g) > p) {
    /* split the interval */
    rateTorqueAndForceEmbedded(T,t_s,(t_s + t_g)/2,τ)
    rateTorqueAndForceEmbedded(T,(t_s + t_g)/2,t_g,τ)
}
else {
    /* calculate force and torque at time t_g (Newton-Euler algorithm) */
    double[] τ^n = M(T^D(t_g))

    /* get the new maximum force and torque for each joint */
    for (int i = 1, i ≤ n, i++)
        if (| τ_i^n |> θ_i^max ∧ (| τ_i^n | −θ_i^max)/θ_i^max > τ_i)
            τ_i = (| τ_i^n | −θ_i^max)/θ_i^max
}
```

The reference point technique that we have introduced here will be used again in Section 7.3 (*Dynamic Collision Test*) on page 87. The approximation works quite well if the movement of each joint inside the interval is monotone in one direction (which is the case most of the time). We want to remark, that the calculation of the above function can only terminate if the following condition holds. The maximal distances between the reference points must approach zero if the time distance approaches zero. This is true if the generated trajectory is derivable in each joint.

6.4 Summary and Outlook

In this chapter, we have shown how the actual force and torque violation in each joint gives a rating for a trajectory. Later in Chapter 8, we will calculate the rating for each trajectory section. Then the force and torque rating for the whole trajectory is the sorted sequence of the section ratings. We call the rating r_t that represents the constraint that forces and torques of the robots are lower than the dynamic limits of the robot *torque limit rating*. In most applications this is a mandatory constraint.

Let us now turn to other aspects of torque and force. For optimisation reasons, it can be interesting to consider the overall energy used in a trajectory. A rating r_e that reflects this is called *energy rating*. Here, the goal is to use as little force and torque in each joint as possible. The rating for a single configuration $\vec{\upsilon}_d$ can be given by

$$r_e^s = \sum_{j=1}^{n} \frac{|\,\tau_j\,|}{\theta_j^{\max}}, \text{ with } \vec{\tau} = M(\vec{\upsilon}_d)$$

As before, the rating of a whole trajectory section T_i is the maximum rating appearing during the respective time interval $[t_{i-1}, t_i]$. This rating normally never becomes zero, since even if the robot does not move, the joints have to hold the links against gravity.

For this reason, the planning algorithm would never terminate if no other termination condition is given than to reach zero in all ratings of each section (see Section 8 for a detailed discussion).

Another desired optimisation could be to keep torques and forces evenly distributed over time, that is, the used force and torque should be identical in all trajectory sections. This means the deviation from the mean average should be zero. Let the number of base points be m. Then the *mean average* rating could be given as follows.

$$r_a = \left| \hat{r}_i - \frac{\sum_{i=1}^{m} \hat{r}_i}{m} \right|$$

with

$$\hat{r}_i = \max \left(\sum_{j=1}^{n} \frac{\mid \tau_j \mid}{\theta_j^{\max}}, \text{with } \vec{\tau} = M(T^D(t)), t_{i-1} < t < t_i \right)$$

This rating considers all trajectory sections and tries to decrease the distance to the mean average of all sections.

Another, quite different rating is the *time rating* which tries to minimise the time that it takes the robot to move from start to goal. This usually fully exploits the dynamic limits of at least one joint of the robot. However, a rating maximising the used force and torque is not helpful, as the robot would try to move as much as possible between the start and goal time point, but not straight towards the goal configuration. A better strategy is to force the robot to use all the capacity it has quite early, in order to reach the goal as quickly as possible. In order to force the exact trajectory to reach the goal configuration at an early point in time, we need a trajectory of at least four base points and the last three base points need to have the same position as the goal configuration. This forces the exact trajectory to reach the goal configuration as early as the base point before the last base point and to stand still in the last trajectory section. To this end, we suggest two ratings for the last three base points $\vec{b}_{m-2}$, $\vec{b}_{m-1}$, and $\vec{b}_m$ as follows

$$\begin{aligned} r_{m,1} &= \sum_{j=1}^{n} \mid \omega_{m,j} - \omega_{m-1,j} \mid + \sum_{j=1}^{n} \mid \omega_{m-1,j} - \omega_{m-2,j} \mid \\ r_{m,2} &= \frac{1}{t_m - t_{m-1}} \end{aligned}$$

Minimising the rating $r_{m,1}$ means minimising the distance of the base points at time points $t_m = t_g$, t_{m-1}, and t_{m-2}. The rating $r_{m,2}$ is maximising the time elapsed between the last two configurations. As a consequence, the robot tries to reach the last but one base point as early as possible. Simultaneously, the robot's configuration at this base point will be quite close to the goal configuration. The combination (e.g. sum) of both ratings should result in the desired behaviour. Note that this rating is the same for all trajectory sections.

Finally, we would like to mention that besides criteria related to torque and force, many other quality criteria of trajectories exist, that can be rated and optimised. For example, one might be interested in finding a short trajectory. The length of a trajectory may be reduced in configuration space or in the real world. A short path in the configuration space does not necessarily imply a short trajectory of the links in physical space.

The easiest way is to approximate the movement in the configuration space or the movement of a link's coordinate system by linear movements. Then the overall distance can be calculated and minimised.

CHAPTER 7

Collision Rating

A static collision test is given that not only detects collisions but also computes a rating reflecting collision depth. The test is extended to a dynamic collision test which is able to check for collisions along a given time interval. We then show how the dynamic collision test can be used to rate a complete trajectory. Finally, we summarise our results and discuss extensions.

7.1 Introduction

One of the main ingredients for our motion planning algorithm is a reliable and consistent function that rates a given trajectory with regard to collisions. Input to this function is a trajectory, while the output is supposed to be a number indicating its quality. A higher number indicates less quality, that is, deeper collisions, while zero means that the trajectory is free. It is important that our rating function appropriately reflects collision depth in order to guide our planner. Ideally, the function should reflect even the smallest improvements. Efficiency is also crucial, since many intermediate trajectories will need to be rated until our planner is successful.

To speed up collision detection, we test for collisions only at certain points in time. In each collision test a safety distance is incorporated that ensures that if two collision tests at successive points in time report no collision then it is guaranteed that no collision will occur in the interval between those two points in time. The safety distances and the test points must be chosen carefully. If the distances are too large our planner encounters problems when free space is narrow. If they are too small we have to perform many tests or we might overlook collisions.

We solve this problem using an adaptive algorithm. We start with only a few test points and consequently large safety distances. At intervals with no collision everything is fine. At those points where collisions have been detected and where the safety distances exceed a certain limit, we do additional testing at intermediate points in time with smaller safety distances. We continue bisecting affected intervals until the safety distances are sufficiently small.

To further speed up computation, we use two different safety distance functions, an approximate function based on heuristics and a mathematically correct function. The approximate function is used in the beginning and it gives us safety distances that are usually smaller than the correct function but still appropriate in most cases. After we have found an "approximately free" trajectory using the approximate safety distance function, we check for collisions using the correct safety distances. If we find that there are still collisions, we continue our search using only the exact safety distance function from then on. However, our experimental results show that in most cases the trajectory found using the approximate safety distance function is already free.

We have split our discussion of collision testing and rating into three sections. In the next section a *static collision test* is described. The static collision test evaluates the environment at a given time t and configuration $\vec{\upsilon}$ of the robot. We say that the configuration $\vec{\upsilon}$ of the robot at time t is free with distance d if there is no intersection of the robot's triangles with the robot's or the obstacles' triangles. Furthermore the distance of all triangles of one link have at least distance d to all obstacles' and relevant links' triangles. If the configuration of the robot at that time is not free with distance d then it is colliding with a rating $c(\vec{\upsilon},t,d)$. If $c(\vec{\upsilon},t,d) = 0$ then the configuration is free.

In the section thereafter, we discuss the *dynamic collision test*. We use the dynamic collision test to evaluate whether there is any collision during a time interval $[t_1,t_2], t_1 \leq t_2$. During this time the robot may move on a given trajectory section. This is done by performing collision tests at discrete time points over the section. Besides the collision information, the depth or rating of the collision is given, if the configuration or trajectory section collides.

In Section 7.4, we focus on the overall collision rating of a complete trajectory using the dynamic collision test.

7.2 Static Collision Test

A well-known approach to collision testing is to pre-compute geometrical information on the participating objects and to reuse that information in order to speed up each collision test. Geometrical information is gathered by decomposing the objects hierarchically and by finding simple bounding volumes for the objects' components on each decomposition level. The idea is to test for collision using the simple bounding volumes first before testing the more complex shapes of the objects. Since the decomposition is hierarchic, the collision test can be performed in a hierarchic manner as well, starting with the bounding volume on the highest level and descending into deeper levels if a collision has been detected.

Our static collision test is based on the oriented bounding box method of [Gottschalk et al., 1996]. For this collision test it is necessary for the surfaces of the obstacles and the robot to be modelled using triangles. For information on converting surfaces or model representations into a triangle surface representation of the environment we refer to [Fortune, 1992, Barequet et al., 1998]. We give a detailed description of the method of [Gottschalk et al., 1996] in the following. In addition, we describe our extensions that enable us to take safety distances into account and to rate the quality of a collision (that is, its depth).

The facets of the robot and the environment are stored in a pre-computed hierarchical representation of the objects' surfaces using tight-fitting oriented bounding boxes. There is one hierarchical representation $\mathcal{T}_{L_i}$ for each link L_i of the robot, one $\mathcal{T}_S$ for the static environment, and one $\mathcal{T}_{O_j}$ for each time-varying obstacle O_j. The hierarchical representation of the object's surface is a binary tree, with each node representing a box in the three dimensional physical space.

In Figure 7.1, a two dimensional example for hierarchical placement of oriented bounding boxes around triangles is given. From left to right and top to bottom, the group of triangles is divided hierarchically and new boxes are found. The children (e.g. $\mathcal{B}_{00}$ and $\mathcal{B}_{01}$) are two boxes dividing the triangles in the father's box $\mathcal{B}_0$ into two groups.

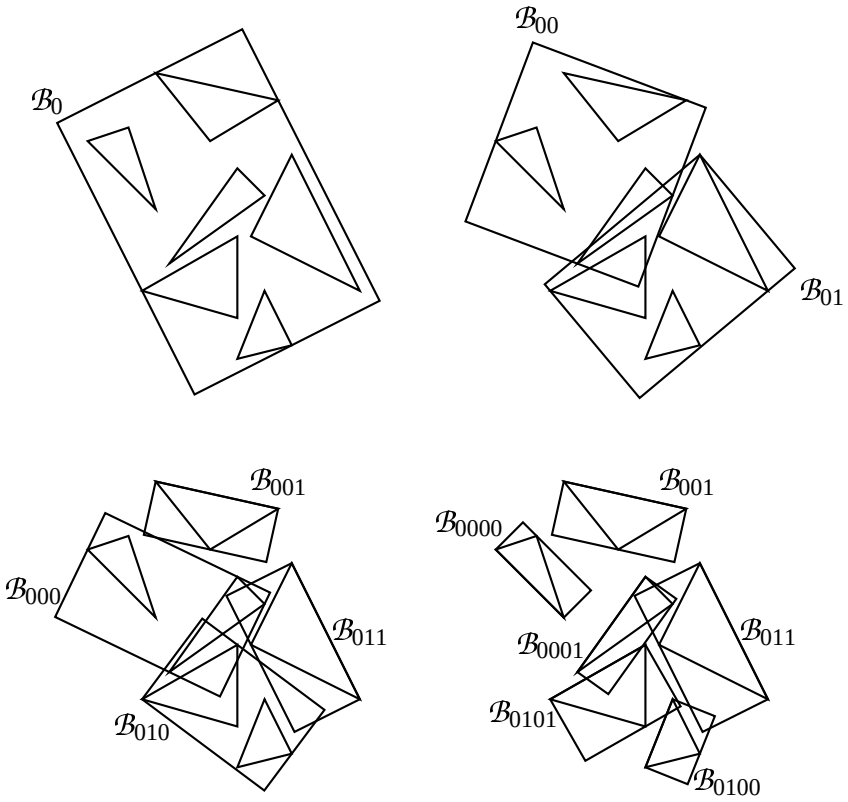

Figure 7.1: *Two dimensional example for hierarchical box placement with six triangles.*

In Figure 7.2 the corresponding tree is shown. The root contains all triangles of the object. A leaf contains only one triangle. The approach of the oriented bounding box method is to find tight-fitting bounding boxes around groups of triangles and to divide the triangles into two groups such that the volumes of the boxes are getting smaller very quickly [Gottschalk et al., 1996].

The main part of the collision test is now to test two oriented bounding boxes (of different trees) against each other, and to decide whether they collide. A box $\mathcal{B}$ is given by its centre $\vec{o}_{\mathcal{B}}$, three orthonormal unit vectors $\vec{v}_{\mathcal{B},i}$ and three extensions $e_{\mathcal{B},i}$, $i = 1,2,3$.

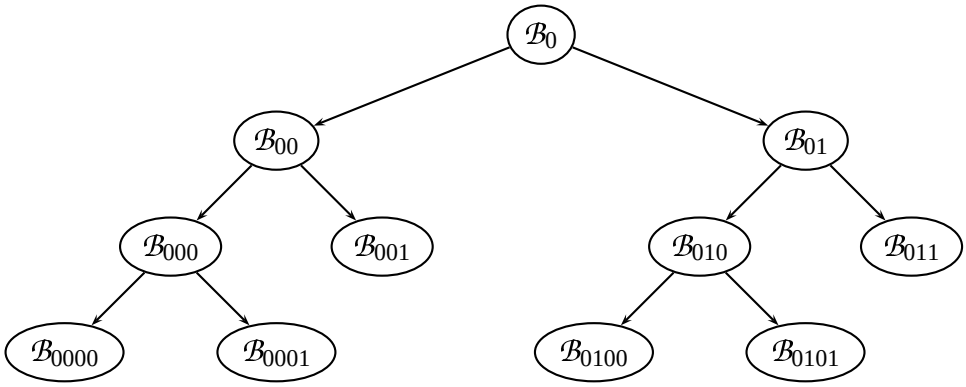

Figure 7.2: *Hierarchical tree of the example above. Each node represents one oriented bounding box.*

The eight vertices of the box $\mathcal{B}$ are

$$\vec{o}_{\mathcal{B}} + \sum_{i=1}^{3} \pm e_{\mathcal{B},i} \vec{v}_{\mathcal{B},i}$$

We add a safety distance d by either enlarging each extension of one box by d or enlarging the extensions of both boxes taking part in the collision test by $\frac{d}{2}$. For simplicity, we look at an example where the whole safety distance is added to one box. For the collision test, the centres and the radii of the boxes have to be projected onto a vector $\vec{L}$ (see Figure 7.3). The choice of $\vec{L}$ will be explained later.

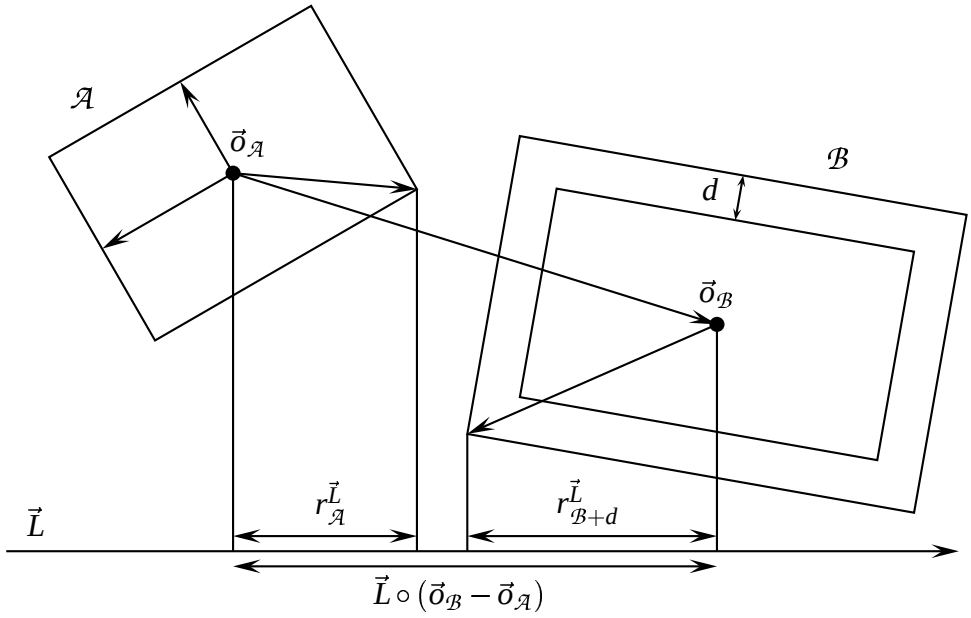

Figure 7.3: *Projection of oriented bounding box onto a vector.*

In the three dimensional case the length of the radii of a box $\mathcal{B}$ projected on a vector $\vec{L}$ is calculated as follows.

$$r_{\mathcal{B}}^{\vec{L}} = \sum_{i=1}^{3} \left| \vec{L} \circ \left(e_{\mathcal{B},i} \vec{v}_{\mathcal{B},i} \right) \right|$$

If there is a safety distance d around a box, as demonstrated for box $\mathcal{B}$, the length of the radii of a box $\mathcal{B}$ is

$$r_{\mathcal{B}+d}^{\vec{L}} = \sum_{i=1}^{3} \left| \vec{L} \circ \left(\left(d + e_{\mathcal{B},i} \right) \vec{v}_{\mathcal{B},i} \right) \right|$$

Let $\mathcal{A}$ be an oriented bounding box somewhere in the hierarchical representation of a robot's link $\mathcal{T}_{L_i}$ and let $\mathcal{B}$ be an oriented bounding box somewhere in the hierarchical representation of an obstacle $\mathcal{T}_{O_j}$. The two boxes $\mathcal{A}$ and $\mathcal{B}$ do not intersect and have safety distance d (see Figure 7.3) if we find a projection vector $\vec{L}$ such that

$$\vec{L} \circ (\vec{o}_{\mathcal{B}} - \vec{o}_{\mathcal{A}}) > r_{\mathcal{A}}^{\vec{L}} + r_{\mathcal{B}+d}^{\vec{L}}.$$

This means that if we find a projection direction where the centres have a farer distance than the sum of the radii (and the safety distance) of the boxes, then the boxes do not intersect and have at least the given safety distance.

The vector $\vec{L}$ is chosen such that all relative positions of the two boxes are covered. Therefore the projection test has to be done fifteen times with different projection vectors $\vec{L}$, trying to find one which satisfies the above condition. It has to be repeated six times with the box axes and nine times with the cross product of the box axes. The first six projections (three possibilities per box) catch the cases, where the boxes are lying on different sides of a plane parallel to one side of a box. In the other cases the boxes have to lie on different sides of a plane which is parallel to two edges of the boxes. If there is any projection which satisfies the inequation, then the boxes do not intersect and have at least safety distance d. Beside the boxes, the triangles inside the boxes have the same safety distance. If no projection is found that satisfies the inequation, then the two boxes collide or do not have the safety distance d. In this case, each child of the larger box is tested against the smaller box. If there are no children left in the larger box, then the children of the smaller box are tested against the larger box. Both tests have to fulfil the inequation or the splitting is repeated.

At the bottom of the hierarchy if there are no children left in either box, then both boxes contain a single triangle. Let a be a triangle in a leaf of $\mathcal{T}_{L_j}$ and let b be a triangle in a leaf of $\mathcal{T}_{O_i}$. Furthermore, let $\vec{v}_{a,1}, \vec{v}_{a,2}, \vec{v}_{a,3}$ be the vertices of triangle a and let $\vec{v}_{b,1}, \vec{v}_{b,2}, \vec{v}_{b,3}$ be the vertices of triangle b. These triangles have to be tested against each other.

The triangle test is done almost the same way as the box test. There are seventeen projections necessary to secure freedom from collision and safety distance. The projections are twice on the normals of the triangles, nine times on the cross product of the triangle sides, and six times on the cross product of the normal and the edges of the triangles. The first two tests cover the case where one triangle lies outside a plane through the other triangle. The next nine projections test whether there is a plane parallel to the edges of

the triangles separating them. The last six tests are important for the case of two triangles in the same plane. They check whether a plane in one edge of the triangle and vertical to the triangle is separating the other triangle. In Figure 7.4, an example for a projection on the cross product of the normal of triangle a and the edge $\vec{v}_{a,1} - \vec{v}_{a,3}$ is depicted (one of the last six cases).

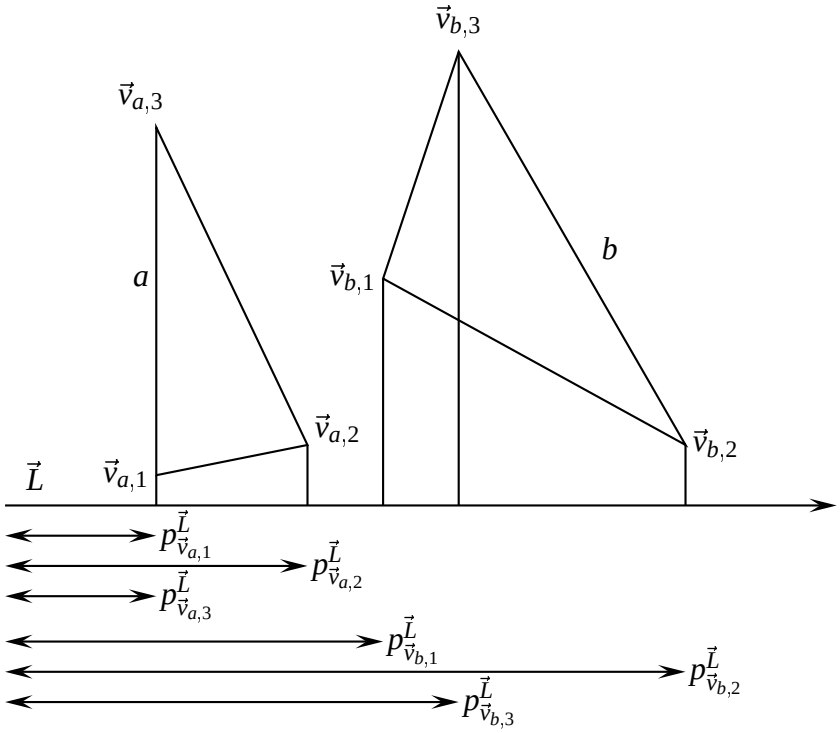

Figure 7.4: *Projection of two triangles onto a vector.*

Each vertex of each triangle is projected on the projection vector $\vec{L}$.

$$p^{\vec{L}}_{\vec{v}_{t,i}} = \frac{\vec{L} \circ \vec{v}_{t,i}}{|\vec{L}|}, \text{ with } t = a,b \text{ and } i = 1,2,3$$

To secure a distance larger than d, the projection values $p^{\vec{L}}_{\vec{v}_{t,i}}$ have to be compared. All projection values of one triangle have to be more than d less the values of the other triangle.

$$\max\left(p^{\vec{L}}_{\vec{v}_{t_1,i}} \mid i = 1,2,3\right) + d \leq \min\left(p^{\vec{L}}_{\vec{v}_{t_2,i}} \mid i = 1,2,3\right)$$

If the above equation holds for either $t_1 = a, t_2 = b$ or $t_1 = b, t_2 = a$, then the triangles a and b have at least distance d. It is obvious, that the greatest distance of all projections give a lower bound for the real distance $d_r(a,b)$ of the triangles a and b,

$$\begin{aligned} d_r(a,b) &= \max\left(\min\left(p^{\vec{L}_j}_{\vec{v}_{t_2,i}} \mid i = 1,2,3\right) - \max\left(p^{\vec{L}_j}_{\vec{v}_{t_1,i}} \mid i = 1,2,3\right)\right) \\ &\quad \text{with } t_1, t_2 = a,b \text{ and } \vec{L}_j, j = 1,\ldots,15 \end{aligned}$$

If two triangles are too close to each other, we want to give a rating for the collision depth depending on a collision centre $\vec{o}$ of the link (see Figure 7.5). The choice of the link's collision centre $\vec{o}$ will be described later.

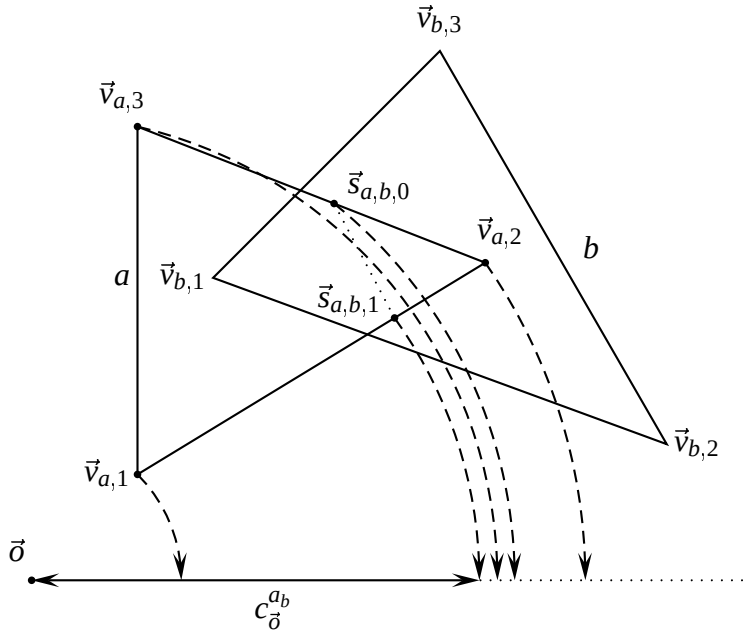

Figure 7.5: *Rating the amount of collision of two triangles.*

The vector $\vec{o}$ gives the centre of the collision rating of one triangle, this means that closer collisions are worse. We have to distinguish between intersecting triangles and triangles that are just too close to each other. If there is an intersection, then first the intersection segment $\left[s_{a,b,0}, s_{a,b,1}\right]$ of the two triangles a and b is calculated. The distance of the vertices $\vec{v}_{a,1}$, $\vec{v}_{a,2}$, and $\vec{v}_{a,3}$ of triangle a and the vectors $\vec{s}_{a,b,0}$ and $\vec{s}_{a,b,1}$ from the centre $\vec{o}$ are calculated. If the triangle cannot satisfy the safety distance, then the collision distance $c(a,b)_{\vec{o}}$ of the triangle a with the triangle b based on the collision centre $\vec{o}$ is defined as follows.

$$c_{\vec{o}}(a,b) = \begin{cases} \min\left(|\vec{s}_{a,b,i} - \vec{o}| \mid i = 0,1\right) & : \quad \text{if } \exists \vec{s}_{a,b,0}, \vec{s}_{a,b,1} \\ \max\left(|\vec{v}_{a,i} - \vec{o}| \mid i = 1,2,3\right) + d_r(a,b) & : \quad \text{if } d_r^{a_b} \leq d \\ \infty & : \quad \text{if } d_r(a,b) > d \end{cases}$$

If there is no intersection between the two triangles, then the closest distance $d_r(a,b)$ between the two triangles is used to calculate the collision value of the triangle. The nearest distance $d_r(a,b)$ is the maximum distance which can be determined during the projections. If this distance is larger than the safety distance, then there is no collision and the collision distance is infinite. We can now give the collision distance of a triangle a in the whole environment based on the centre $\vec{o}$.

$$c_{\vec{o}}^{a} = \min\left(c_{\vec{o}}(a,b) \mid b \neq a, b \text{ triangle in environment}\right)$$

The next step is to give a rating for each link L_l of the robot. For this we choose the centre $\vec{o}_l$ based on the connecting joint J_l and the previous link $L_{p(l)}$, so each link of the robot has its own collision centre $\vec{o}_l$. The centre $\vec{o}_0$ of the base link can be chosen arbitrarily, since we ignore the rating of the base link because it cannot be moved. In addition we assume that the base is collision free for the whole planning process. The position of the collision centre $\vec{o}_l$ of the other links should be close to the important triangles and far from the less important triangles of the link L_l. For a rotational joint the triangles nearest to rotation axis are more important, since it takes more effort to move a triangle nearer to the motion axis. If the connection is via a translational joint, then no triangles have to be preferred because of the joint's type. If we look at the kinematic structure of the robot and therefore at the previous link of link L_l, we can make another observation. The triangles which are closer to the previous link are more important. The problem is that the previous link $L_{p(l)}$ changes its position in relation to the link L_l. For this we take the point of contact with the previous joint in the real world as collision centre. This point has to be given with the description of the robot. Similar considerations can be found in [Baginski, 1999] in connection with the "shrink centre".

The overall rating of a link L_l is based on the centre $\vec{o}_l$ and returns the absolute depth of the collision. Let a be a triangle of the link L_l and let b be a triangle of the environment minus the link itself and its following links. Moreover, let r_l be the maximal extension of the link $r_l = \max\left(|\vec{v}_{a,i} - \vec{o}_l| \mid a \in L_l, i = 1,2,3\right)$

$$C_l = \begin{cases} r_l + d - \min\left(c_{\vec{o}_l}(a,b) \mid a \in L_l, b \in \mathcal{L} \setminus \mathcal{L}_l^{\mathbf{nl}} \cup O\right) & : \quad \text{if } \exists \vec{s}_{a,b,0}, \vec{s}_{a,b,1} \\ d - \min\left(|d_r(a,b)| \mid a \in L_l, b \in \mathcal{L} \setminus \mathcal{L}_l^{\mathbf{nl}} \cup O\right) & : \quad \text{if } \exists b : d_r(a,b) \leq d \\ 0 & : \quad \text{if } \forall b : d_r(a,b) > d \end{cases}$$

This means that if any triangle in the link collides with the considered environment, then the collision value is the radial distance between the intersection which is closest to the centre and the maximal extension of the link plus the safety distance. If the safety distance is not large enough, then the violation part of the safety distance is taken. Otherwise the collision value is zero.

During the calculation of C_l it is important to consider the triangles of the other links apart from the triangles of L_l too, in order to detect a self collision of the robot. All links except the next links in $\mathcal{L}_l^{\mathbf{nl}}$ have to be considered. This represents the state of a link. A link nearer to the base is less responsible for a collision than its children. In practice, one would even exclude the neighbouring links from the calculation, firstly because of inaccuracies introduced during modelling and secondly because of the safety distance we need during the dynamic collision test. (It is useful to add the decision whether two links should be tested against each other to the given information of the model.)

The rating of a link L_l that also considers the topology of the robot is defined recursively as follows.

$$\hat{C}_l = \begin{cases} 0 & : \quad \text{if } l = 0 \\ C_l & : \quad \text{if } \hat{C}_{p(l)} = 0 \\ r_l + d & : \quad \text{if } \hat{C}_{p(l)} \neq 0 \end{cases}$$

This means that a link collides completely if a previous link collides. Therefore the whole rating of the robot in position $\vec{v}$ and all obstacles at time t with safety distance d is

$$C(\vec{v},t,d) = \sum_{L_l \in \mathcal{L}} \hat{C}_l$$

This results in a collision test starting with the links nearer to the robot's base, as the value $C_{p(l)}$ of the previous link has to be known. As we can save the minimal and maximal collision value of all triangles in each bounding box hierarchically for the links of the robot, it is possible to select the box with the worst possible collision value first, so we can make sure that we get the maximal collision value quite quickly. On the other hand, the determination of the collision value can be stopped if the triangles in a box cannot worsen the rating. Figure 7.6 shows the time t (in milliseconds) necessary for a collision test with rating (black plot) and without rating (grey plot) using different safety distances d (in millimetres) on the left side. On the right side the percentage of collisions c is printed over the safety distance d. Clearly the curves with rating and without rating have to be equal. For safety distances larger than approximately 130 all tests result in collisions. The times of the collision test without rating and safety distance zero are similar to the original ones. The only difference is that there are a few more additions in the box and triangle test.

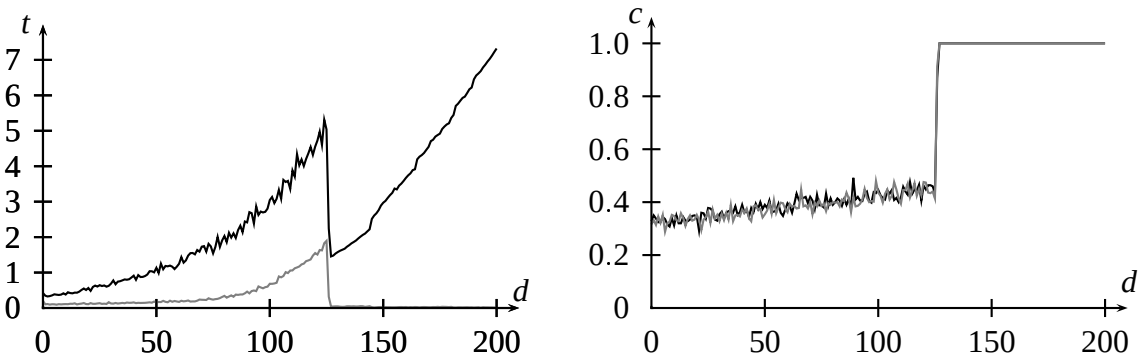

Figure 7.6: *Comparing collision test: 500 tests per safety distance, 6504 obstacle triangles, 2494 robot triangles, approximately 30% of the tests result in collisions if safety distance is zero.*

We note that the times do not vary very much for smaller safety distances, even if the rating is calculated. The other important observation is that the time for a collision test without a rating does not grow as quickly as the collision test with a rating. We will exploit this fact by computing the rating only for small safety distances. As long as safety distances are large, we do not rate. Furthermore the time for a collision test without a rating stops growing when the safety distance is reached, where all tests result in collision.

The static collision test with a rating detects on one hand if the robot has a certain distance d to the obstacles at time t. On the other hand a rating of the time configuration is given. The test cannot detect if an object lies totally inside the robot‘s surface since we test surfaces against each other. But since we assume that the start and the goal configurations are free and in particular are not within an obstacle, it is obvious that if a free trajectory is found, no crossing with an obstacle is possible.

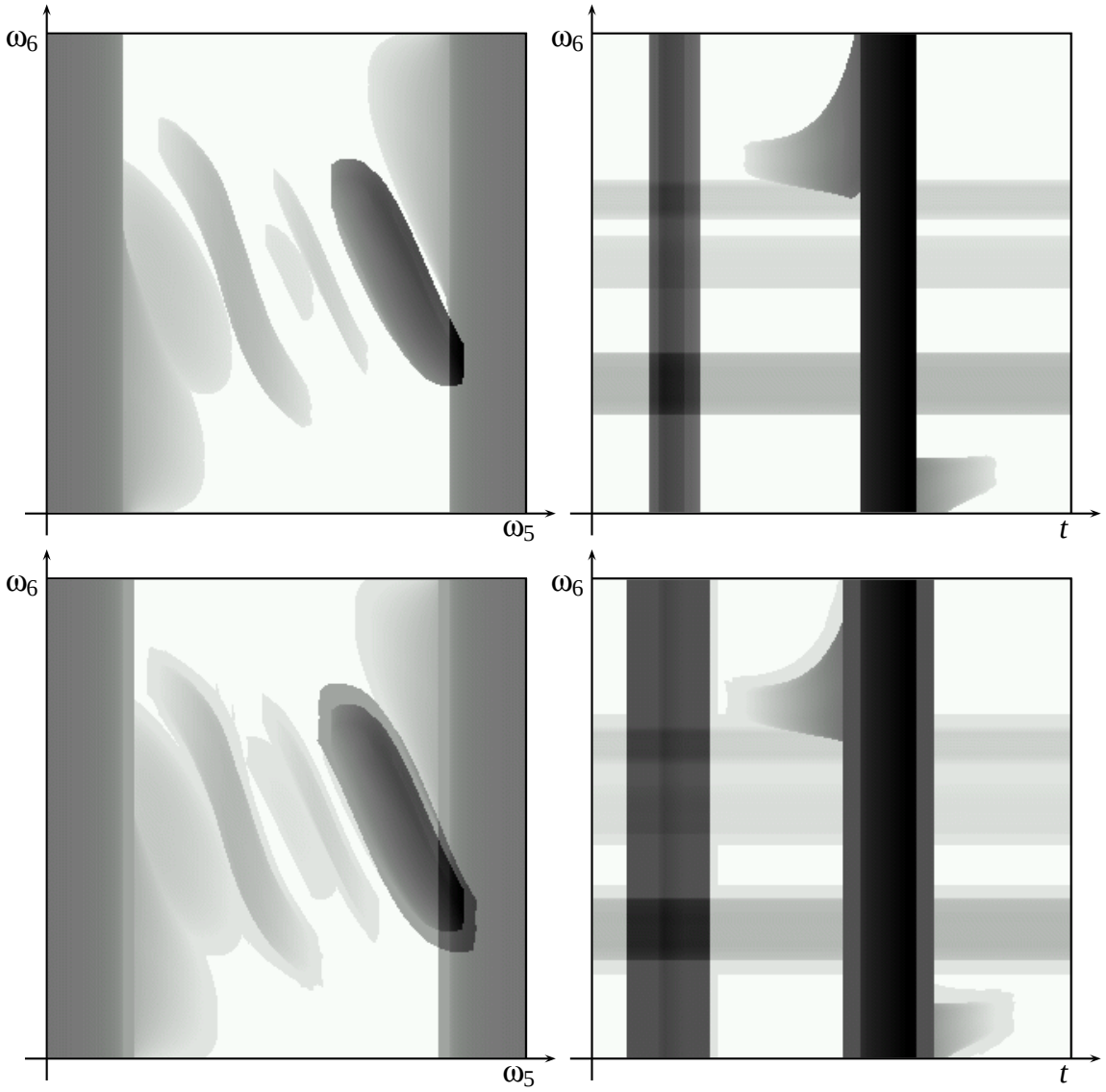

Figure 7.7: *Four examples of the rated configuration space of a 6-dof robot. The upper images show the configuration space without a safety distance, while in the lower images there is a safety distance of* 50*. The configuration space on the left side is the same as in Figure 4.8 and that on the right side is the same as in Figure 4.9.*

In Figure 7.7 the configuration space for the well-known six degrees of freedom robot is depicted (see Figure 4.1 on page 28 for a description of the robot). Only the joints J_5 and J_6 are moved in the left pictures and J_6 and the time is changed in the right pictures. The two configuration spaces are the same as in Figure 4.8 on page 39 and in Figure 4.9 on page 40, but now we rate the collisions. The darker a region is, the worse the configuration of the robot is. The rating is done without any safety distance in the upper configuration spaces and with a safety distance of 50 in the lower pictures.

7.3 Dynamic Collision Test

We are given a time interval and a trajectory. Our goal is to detect a collision during the time interval by performing only one static collision test. During that interval, the robot is moving according to the given trajectory and the time-varying obstacles are moving as well. We have to determine a point in time within the interval and safety distances that are large enough such that when our static collision test detects no collision, then it is guaranteed that no collision is present along the whole time interval.

On the other hand, we have to accept that it might be a false alarm if a collision is detected by our static test. This is simply because our safety distances are so large. After all, there must be provision for movements of the robot and the obstacles within the given time interval. In the case that the static test reports a collision and the current safety distances exceed a certain limit (in other words, the precision of our collision test is too low) we will bisect the interval and repeat the test on the resulting intervals. We defer the description of this adaption process to the next section. In this section we will concentrate on a single interval and we will show how a suitable point in time and appropriate safety distances can be found for the static collision test.

Work on safety distance computation in static environments can be found in [Glavina, 1991] and [Baginski, 1999]. In [Glavina, 1991], a safety distance is applied to the static obstacles, while no safety distance is applied to the robot. In [Baginski, 1999], safety distance is achieved by blowing up the robot's geometry. In time-varying environments, however, it turns out to be advantageous to have distinct safety distances for the robot and for the time varying obstacles. This way the safety distances can be kept lower than in the case where only the robot or the obstacles have a safety distance. In particular, static obstacles do not have safety distances at all.

The time interval which has to be tested starts at t_s and ends at t_g. Firstly, we have to calculate an interval $[\vec{\upsilon}_b, \vec{\upsilon}_t]$ which bounds the positions of the robot joints in the given trajectory section. Let $\upsilon_{b,l}$ and $\upsilon_{t,l}$ be the minimal and maximal limits of joint J_l. This information can either be provided by the trajectory generation algorithm or the allowed area described via the base points can be taken. However, if we are still at the beginning of the planning process with only a small number of base points and the bound cannot be given by the trajectory generation, this is only a rough estimate and will lead to a large safety distance. Consequently, it is likely that the algorithm reports collisions that do not actually exist. At the end of this section we describe an approximation method that avoids this problem.

In addition, a speed vector $\vec{\sigma}_{O_i} = (\sigma_{O_i,1}, \sigma_{O_i,2}, \sigma_{O_i,3}, \sigma_{O_i,4}, \sigma_{O_i,5}, \sigma_{O_i,6})^T$ for each obstacle O_i is given which specifies the minimal and maximal position and orientation speed for the time-varying objects (see Section 4.2). Let e_{O_i} be the maximal distance any point of the obstacle has to the origin $o_i(t)$ of the obstacle.

First, we calculate the safety distance for the time-varying objects. The maximal distance a point of a time-varying object can reach (see Figure 7.8) depends on the maximal translation and rotation the object can perform during a time interval.

We look at the time-varying object at time $t_s + \frac{t_g - t_s}{2} = t_s + \triangle t$. If the origin $o_i(t)$ moves with maximal or minimal speed in y-direction (see Figure 7.8), the maximal distance the

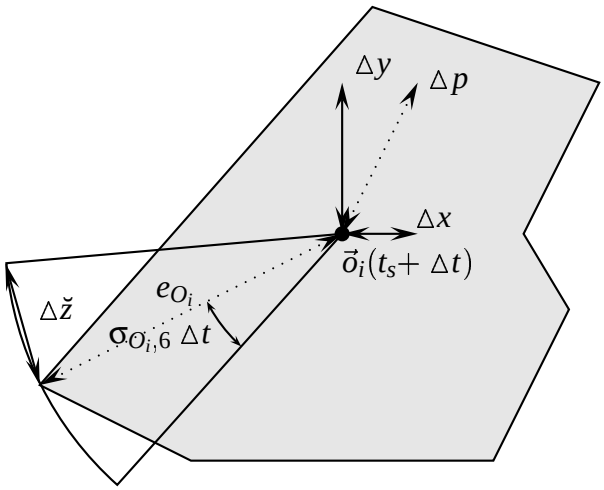

Figure 7.8: *Safety distance for a time-varying object.*

point can move is

$$\triangle y = \sigma_{O_i,2}\,\triangle t$$

The same distances can be given for the x and z direction. The overall distance a point can cover if there is no rotation is

$$\triangle p_i = \left| \begin{pmatrix} \triangle x \\ \triangle y \\ \triangle z \end{pmatrix} \right| = \left| \begin{pmatrix} \sigma_{O_i,1} \\ \sigma_{O_i,2} \\ \sigma_{O_i,3} \end{pmatrix} \right| \triangle t = \sqrt{\sigma_{O_i,1}^2 + \sigma_{O_i,2}^2 + \sigma_{O_i,3}^2}\,\triangle t$$

Let us next take a look at the rotational case. In the following, let $\cos(x)$ be -1 if the value of x is greater than π or smaller than $-\pi$. For the rotation (see Figure 7.8) we get a maximal distance $\triangle \check{z}$ if there is a rotation around the z axis, of

$$\triangle \check{z} = \sqrt{sin(\sigma_{O_i,6}\,\triangle t)^2 + (1 - cos(\sigma_{O_i,6}\,\triangle t))^2}\, e_{O_i} = \sqrt{2(1-\cos(\sigma_{O_i,6}\,\triangle t))}\, e_{O_i}$$

where e_{O_i} is the furthest point of the object from the origin. The overall distance a point can cover if there is no translation is

$$\triangle o_i \;=\; \sqrt{2(1-\cos(\sqrt{\sigma_{O_i,4}^2 + \sigma_{O_i,5}^2 + \sigma_{O_i,6}^2}\,\triangle t))}\, e_{O_i}$$

So we can be sure that there is no collision with a time-varying object between t_s and t_g and the robot does not move if the collision rating $C(\vec{v}_b, t_s+\triangle t, \max(\triangle p_i + \triangle o_i))$ with safety distance $\max(\triangle p_i + \triangle o_i)$ is zero with the time-varying obstacle placed at time $(t_s+t_g)/2$. This means that the following inequation has to hold for each obstacle O_i

$$\begin{aligned} d &\geq \triangle p_i + \triangle o_i \\ d &\geq \sigma_{p_i}\,\triangle t + \sqrt{2(1-\cos(\sigma_{o_i}\,\triangle t))}\, e_{O_i} \\ &\quad \text{with } \sigma_{p_i} = \sqrt{\sigma_{O_i,1}^2 + \sigma_{O_i,2}^2 + \sigma_{O_i,3}^2} \text{ and } \sigma_{o_i} = \sqrt{\sigma_{O_i,4}^2 + \sigma_{O_i,5}^2 + \sigma_{O_i,6}^2} \end{aligned}$$

The next step is to give a safety distance for each link of the robot. The safety distance of the link has to ensure that no point on the link moves further than this distance from a given configuration while the joint values are in a delta interval. We can use the same observations as for the time-varying object. The robot is in the observation configuration $\vec{\upsilon}_b + \frac{\vec{\upsilon}_t - \vec{\upsilon}_b}{2} = \vec{\upsilon}_b + \triangle\vec{\upsilon}$ and the maximal delta the robot joints move are $\triangle\vec{\upsilon}$. In Figure 7.9 the revolution case can be seen on the left side and the prismatic case on the right side. The dashed lines indicate the maximal positions the joint may have. The symbol $\breve{d}_l$ represents the maximal distance any point of the link has from the middle position if the joint is a revolution joint and $\hat{d}_l$ represents this distance for a prismatic joint. Let e_{L_l} be the maximal extension of the link L_l. Since we assume that all the links are connected, this should be the worst distance two joints connected to the same link can have.

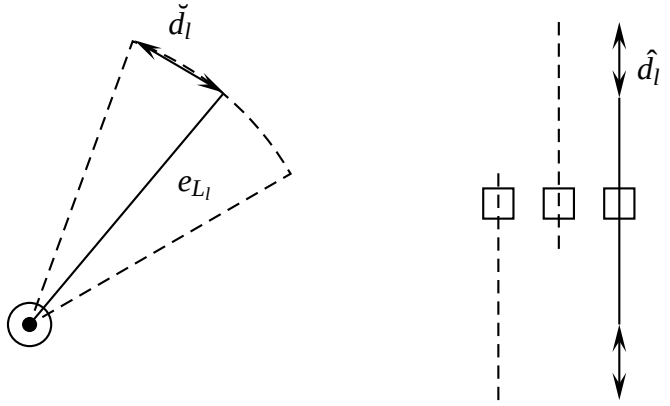

Figure 7.9: *Safety distance for a revolute joint (left) and a prismatic joint (right). Black line indicates middle position, dashed lines extreme positions.*

If the link L_l is connected via a prismatic joint to the previous link, then the maximal distance a point can move from the position $\upsilon_{b,l} + \triangle\upsilon_l$ relative to the coordinate system of this connecting joint J_l is

$$\hat{d}_l = |\triangle\upsilon_l| = \left|\frac{\upsilon_{t,l} - \upsilon_{b,l}}{2}\right|$$

The maximal distance any point inside the link can move for the revolution connection is

$$\breve{d}_l = \sqrt{2\,(1-\cos\,(\triangle\upsilon_l))}e_{L_l} = \sqrt{2\left(1-\cos\left(\frac{\upsilon_{t,l} - \upsilon_{b,l}}{2}\right)\right)}e_{L_l}$$

Both distances are relative to the coordinate system of the connecting joint. Taking both together, the maximal distance d_l a point inside the link L_l can move relative to the origin of the joint J_l is

$$d_l = \lambda_l\breve{d}_l + (1-\lambda_l)\hat{d}_l$$

Now this distance relative to the world coordinate system is needed. To get the safety distance relative to the origin of the robot all previous revolution and prismatic movements

have to be considered. The effect of rotational movement increases the further the links are away from the rotation centre. The overall safety distance relative to the origin of the whole robot can then be calculated as follows, with the maximal position distance $\triangle\upsilon_j = \frac{\upsilon_{t,j}-\upsilon_{b,j}}{2}$ of a joint J_j and the maximal extension e_{L_l} of a link.

$$D_l = \sum_{\forall i:J_i \in \mathcal{J}_l^{\mathbf{pj}}} \left(\lambda_i \left(\sqrt{2\left(1-\cos\left(\triangle\upsilon_i\right)\right)} \sum_{\forall j:J_j \in \mathcal{J}_l^{\mathbf{pj}} \cap \mathcal{J}_i^{\mathbf{nj}}} e_{L_j} \right) + \left(1-\lambda_i\right) \triangle\upsilon_i \right)$$

There is no safety distance for the base. If the previous link is the base, then only the connecting joint to the base is considered. Otherwise the sum of each rotational part has to be put together and multiplied by the overall distance from the joint to the link. This overall distance is the sum of the extensions of all links from the currently observed link and joint respectively to the link for which the safety distance is calculated. Finally the translational safety distance is added.

Let us take the example in Figure 7.10. Let the link's extension be 1 ($e_{L_{1,2}} = 1$). For the left image the lower and upper position bounds of the joints (both revolute) are $[(-30,-30),(30,30)]$ (dashed lines). The set of previous and next joints of a joint are for joint J_1, $\mathcal{J}_1^{\mathbf{pj}} = \{1\}$ and $\mathcal{J}_1^{\mathbf{nj}} = \{1,2\}$ and $\mathcal{J}_2^{\mathbf{pj}} = \{1,2\}$ and $\mathcal{J}_1^{\mathbf{nj}} = \{2\}$ for joint J_2. Therefore the safety distances of link L_1 and L_2 are

$$\begin{aligned} D_1 &= \sqrt{2\left(1-\cos\left(\triangle\upsilon_1\right)\right)} e_{L_1} = \sqrt{2\left(1-\cos\left(30\right)\right)} \\ D_2 &= \sqrt{2\left(1-\cos\left(\triangle\upsilon_1\right)\right)} \left(e_{L_1}+e_{L_2}\right) + \sqrt{2\left(1-\cos\left(\triangle\upsilon_2\right)\right)} e_{L_2} \\ &= \sqrt{2\left(1-\cos\left(30\right)\right)} 2 + \sqrt{2\left(1-\cos\left(30\right)\right)} \end{aligned}$$

The solid lines indicate the test positions of the link. We bisected the problem twice at the second link and the resulting safety distances are depicted in the middle and the right picture of Figure 7.10. For this worst case, where the distance of the joint is equal to the extension and the robot's top moves in an almost circular motion around the base, this distance is quite good.

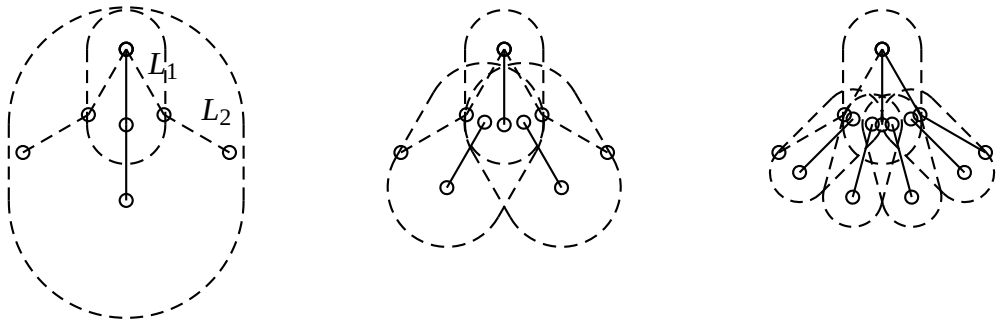

Figure 7.10: *Exact safety distance for a robot (first example).*

In general, the resulting safety distances are a very poor upper limit, especially for robots with lots of joints and different oriented motion axis. In Figure 7.11 another start

and goal position is taken. There it can be seen that the resulting safety distance is much larger than necessary. This also results from the fact that the actual position is not considered.

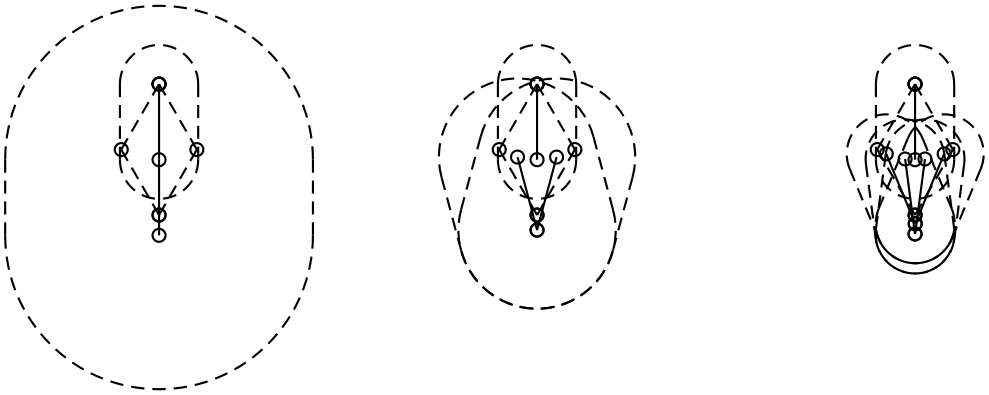

Figure 7.11: *Exact safety distance for a robot (second example).*

Nevertheless, using these exact safety distances, one for each time-varying obstacle and one for each link of the robot, a trajectory can be tested for collision and a collision value can be calculated for a time interval and given joint intervals, by using the static collision test described in Section 7.2. The algorithm for testing a trajectory for collision will be described in the following section.

As we have already noted, the exact safety distances can become quite large for larger intervals. Hence, often a lot of time points in a given trajectory have to be tested to ensure freedom from collision for a given safety distance d, since the time interval has to be bisected if the maximal safety distance for any link is too large. For the trajectory planner, we suggest a two level collision test, one for the rough planning and one for the final collision test.

We use the same approximation as we did for the torque and force rating (see Chapter 6). There the approximation was done by choosing three reference points x_l, y_l, and z_l relative to the origin of J_l for each link L_l, which approximate the size and the position of the link. One point is in the origin $a_l = (0,0,0)^T$, the second gives the expansion

$$b_l = (\max(|v_{t,1}|), \max(|v_{t,2}|), \max(|v_{t,3}|))^T$$

with $t \in L_l$, and the third point $c_l = (-b_{l,1}, -b_{l,2}, b_{l,3})^T$ is for keeping the orientation. If we look at the configuration at t_s and t_g, no point is allowed to move further then $2d$, otherwise the time interval is bisected (see Figure 7.12). The movement itself is determined using the function `getMaxMovement` (see page 72). In the example we assume that the robot moves with constant velocity from the dashed start to goal configuration. The bounds of the interval are then given by the configuration of the robot at t_s and t_g. The advantage of this method is that the actual position of the robot is taken into account and revoking motions are considered. The approximation works quite well if the movement of each joint inside the interval is monotone in one direction (which is the case most of the time).

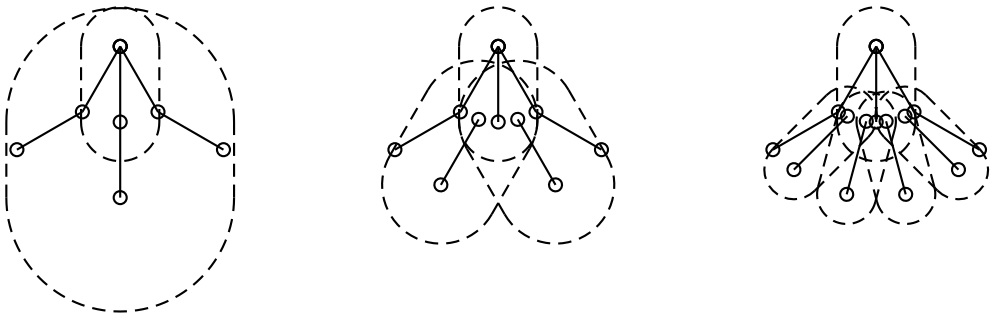

Figure 7.12: *Approximated safety distance for the robot in the first example.*

In our first example, the difference between the approximated and the exact distance is not very large (compare Figure 7.12 with Figure 7.10). In the second example, however, the difference is striking (compare Figure 7.13 with Figure 7.11). Here the advantage of using the current position of the links of the robot can be seen. Especially the approximated safety distance of the second link, which moves in the opposite direction to the first link, is much better than the exact safety distance.

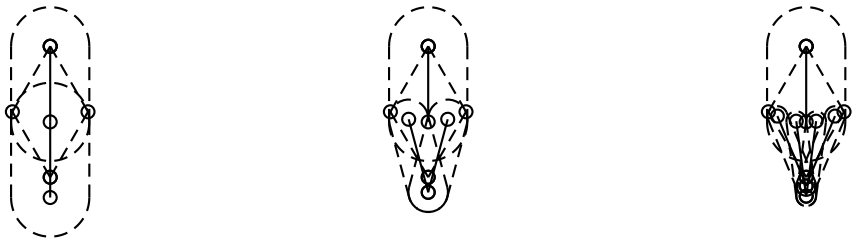

Figure 7.13: *Approximated safety distance for the robot in the second example.*

7.4 Trajectory Collision Rating

In this section we describe the calculation of the collision value of a trajectory $T(t)$ based on the dynamic collision test of the previous section. Let $\hat{T}(t)$ return the robot's configuration in physical space. That is, if $T^D(t) = (\vec{\upsilon}_\omega, \vec{\upsilon}_\sigma, \vec{\upsilon}_\alpha)$, then $\hat{T}(t) = (\vec{\upsilon}_\omega)$. Each trajectory section between two consecutive base points is rated independently with a precision value p representing the safety distance

$$r_c = \max(C(\vec{\upsilon}, t, p) : \vec{\upsilon} = \hat{T}(t), t_{i-1} \leq t \leq t_i)$$

The overall rating of a trajectory section is the sorted sequence of all section ratings (see Chapter 8).

Now we want to give an algorithm for computing the collision rating with a given precision. To this end, let *double* `getSafetyDist` (*trajectory* T, *double* t_s, *double* t_g) be the safety distance which has to be used to secure safety distance for the time interval

$[t_s, t_g]$. For the exact case, this is the maximum of the safety distances for the time-varying obstacle plus the maximum of the safety distances of the robot's links (depending on the lower and upper bounds the robot's joint may reach)

$$d = \max(\triangle p_i + \triangle o_i, i = 0, \ldots, o) + \max(D_l, l = 1, \ldots, n).$$

In the approximated case the calculation of the safety distance is quite simple, as `getSafetyDist` (T,t_s,t_g) = `getMaxMovement` (T,t_s,t_g) /2. As described on page 72, *double* `getMaxMovement` (*trajectory* T, *double* t_s, *double* t_g) returns the maximal distance between two consecutive reference points if the robot is in position $\hat{T}(t_s)$ and $\hat{T}(t_g)$.

Furthermore, the test configuration of the robot has to be calculated to be used for the collision test. Let *double[]* `getTestPosition` (*trajectory* T, *double* t_s, *double* t_g) be the function which returns this robot configuration. For the approximation, this test configuration is simply the median average between the start and the goal configuration $\frac{1}{2}(\hat{T}(t_s) + \hat{T}(t_g))$.

As described in the previous section, it is necessary for the exact collision to be calculated, firstly to get the upper and lower bound of the robot joint values. The test configuration is then the median average between the lower and upper bound. The safety distance can be calculated as we have shown with the dynamic collision test.

The algorithm *double* `rateCollision` (*trajectory* T, *double* t_s, *double* t_g) for calculating the trajectory collision value for section i of a base point trajectory $T(t)$ with precision p (which is equal to the maximal safety distance) is as follows.

---

double `rateCollision` (*trajectory* T, *double* t_s, *double* t_g)
pre-condition: $t_s < t_g$

```
/* calculate the safety distance */
double d = getSafetyDist (T,ts,tg)

/* if safety distance smaller than precision, then return the rated collision value */
if (d ≤ p)
    return C(getTestPosition (T,ts,tg),½(ts+tg),p)

/* if there is no collision, then return */
if (C(getTestPosition (T,ts,tg),½(ts+tg),d) == 0)
    return 0

/* calculate the collision value recursively */
double l,r = 0;
l = rateCollision (T,ts,½(ts+tg))
r = rateCollision (T,½(ts+tg),tg)
if (l ≥ r)
```

```
    return l
else
    return r
```

The idea of the algorithm is to test the whole interval. If there is no collision, then 0 is returned. Otherwise we reduce the time interval, as long as the safety distance is too large. If the actual safety distance is smaller then the given precision p, then the position is rated.

The above algorithm is simple, but there are a number of optimisations that can be employed. First, the rating of the collision should only be done if the safety distance is below the precision. As we have shown in Figure 7.6, the collision test with rating is time consuming, particularly for larger safety distances. Second, it is not necessary to consider all links in all collision tests. As as soon as a link is collision free in a collision test, the link does not have to be tested again in the recursive calls. This happens quite often since the safety distance for links nearer to the base is smaller. This fact also leads to the third optimisation. The safety distances can be different for each link and each time-varying obstacle. Hence, in practice the `getSafetyDist` function should return one safety distance for each link and one for each time varying obstacle.

7.5 Summary and Outlook

We have demonstrated how the oriented bounding box collision test of [Gottschalk et al., 1996] can be modified in order to rate a configuration of the robot regarding collision depth. We then extended the static collision test to a dynamic collision test for time-varying environments. There we suggested using different methods to compute safety distances. An approximate method for the coarse planning phase and an exact method for the final validation. Finally, we showed how an overall rating of a complete trajectory can be calculated.

Our collision rating algorithm has major advantages compared to existing approaches. First of all, we are able to rate the configuration of a robot independent of the shape of the robot and the obstacles. For example, since we do not change the shape or positions of the triangles we do not require the participating objects to be convex (a common prerequisite in existing approaches). Apart from this, the distance collision test (without rating) only costs slightly more then the normal collision test with the oriented bounding box method [Gottschalk et al., 1996]. This is quite important, as we need the distance collision test for our dynamic collision test.

Let us discuss some extensions. It may be of interest to introduce weighted collision rating sums, with weights dependent on the type of the objects participating in a collision. For example, it is reasonable to assume that a collision with a time-varying obstacle is not as bad as a collision with a static obstacle. Let $C^s(\vec{\mathfrak{v}},t,p)$ be the collision value with the static environment and $C^t(\vec{\mathfrak{v}},t,p)$ be the collision value with the time-varying environment. To reflect this, the rating could be given as

$$r_s = \max(C^s(\vec{\mathfrak{v}},t,p) + cC^t(\vec{\mathfrak{v}},t,p) : \vec{\mathfrak{v}} = \hat{T}(t), t_{i-1} \leq t \leq t_i, 0 < c < 1)$$

This *collision type rating* can be used instead of a simple *collision rating*. As before, to have a free trajectory this rating has to be zero.

Another idea is to rate the collision depending on the time when the collision happens (*collision time rating*). Here we assume that an early section in collision is worse then a collision that occurred later in time. A suitable rating would be

$$r_v = \frac{1}{t_i} \max(C(\vec{\mathfrak{v}}, t, p) : \vec{\mathfrak{v}} = \hat{T}(t), t_{i-1} \leq t \leq t_i)$$

Such a rating would be useful if the robot must start to move before the whole trajectory is planned. However in addition, if the robot starts its movement before the planning process has finished, the part of the trajectory that is already executed by the robot must not be further changed by the planning process.

CHAPTER 8

Trajectory Planning

We discuss different aspects of a good trajectory. Thereafter the rating of a whole trajectory by combining these different aspects is given. In the second section, we present and analyse techniques for improving trajectory quality by changing the trajectory locally. There a three main techniques, namely moving, adding, and deleting base points. These techniques are then combined into a planning algorithm that is able to plan trajectories in time-varying environments. In the last section, we analyse the algorithm and discuss its properties.

8.1 Global Trajectory Rating

Let us first focus on some general properties of trajectories. First of all, we have to distinguish between free and unfree trajectories.

Definition 8.1 Free Trajectory – Unfree Trajectory
We say a trajectory is free if there is no collision at any point of the trajectory and all kino-dynamic limits of the robot are fulfilled. If one of those conditions is violated, then we speak of an unfree trajectory.

Based on this notion, trajectories can be evaluated as follows.

- **Free Trajectory** – If the trajectory is free then it is interesting to know more about the quality of the trajectory. For example, how far the objects in the environment are away from the trajectory. Furthermore, it is interesting to know how near the robot gets to the dynamic limits of its joints. Another interesting issue is the comparison of the trajectory to a time or energy optimal trajectory. Moreover, we might want to have an even load on the robot's joints, to spare the hardware of the robot.

- **Unfree Trajectory** – If a trajectory is unfree, then the depth of the collision with obstacles is interesting. If the collision value is higher, then the collision is deeper

and not as good as a lower collision value. Furthermore, we want to know if the collision is with a static or a time-varying obstacle, because a collision with a moving object does not seem as bad as a collision with a static object. An earlier collision is worse than a later one, if we think of a real-time planner, where the robot could start moving as soon as earlier sections of the trajectory are free. Moreover, we might want to rate a trajectory with regard to the amount of the trajectory that is in collision as it might take more time to find a solution when starting with a trajectory which is unfree at many points in time. The second group are the kinodynamic limits of the robot. It is important to know how far the limits are violated. If the violation is small, it might be reasonable to presume that the trajectory is nearer to a free trajectory.

In Chapter 6 (Torque Rating) and Chapter 7 (Collision Rating), we have presented algorithms that calculate the ratings for two of the above aspects. First, a rating of collision depth (r_c) and second, a rating of the violation of dynamic limits of the robot (r_t). Since our main goal is to find a free trajectory for a given start and goal time configuration, not all of the above aspects are equally important. Our main concern is to get a trajectory without a collision and no violation of dynamic limits. If both constraints are fulfilled, then the trajectory is a free trajectory. Therefore, we concentrate on these two constraints in the following, but all other constraints or aspects can be added to the planner by extending the overall rating function.

The simplest combination of ratings, without distinguishing between a free and an unfree trajectory, is adding all ratings r_j powered by e_j and multiplied by a weight f_j. Both values have to be given as they are part of the rating. The weight is necessary to put each rating in the right proportion with all other ratings. The power e_j is for bringing different gradients of the ratings in a useful proportion. The selection of the two values f_j and e_j could be up to the designer of the ratings as well, but sometimes it is useful to change these values according to the problem. For example, if a scene has no obstacles it could be useful to lower the influence of this rating, because only self-collision takes place. Alternatively consider a robot without limited dynamics moving in an environment without time-varying obstacles. In such a setting, it could be useful to lower the influence of the torque rating. In the following, let $r_{i,j}$ be the rating r_j of the trajectory section between t_{i-1} and t_i.

A rating function for some criteria should have the following properties. First, the value of the rating has to represent the actual condition of the constraint in a trajectory as well as possible. The rating should have as few local minima as possible. We assume that the constraint that we rate is better fulfilled if the rating for the trajectory delivers *lower* values. Each rating has to be equal or greater than zero.

Definition 8.2 Simple Rating
Let T be a trajectory with $m+1$ base points. For all ratings $r_j, 0 \leq j \leq n$ let $f_j \geq 0$ and $e_j \geq 0$. The simple rating of section i is given by

$$R_i = \sum_{j=0}^{n} f_j r_{i,j}^{e_j}$$

The overall simple rating R for the whole trajectory is a sorted sequence of all trajectory section ratings R_i which are not zero ($R_i \neq 0$). Let $s(i)$ be a bijective function with a range and image of $1 \leq i \leq m$.

$$R^s = \langle R_{s(1)}, \ldots, R_{s(l)} \rangle \quad , \quad \begin{array}{l} \textit{with } R_{s(i)} \geq R_{s(j)} \textit{ if } i < j \\ \textit{and } \forall i \leq l : R_{s(i)} \neq 0 \textit{ and } \forall i > l : R_{s(i)} = 0 \end{array}$$

(If all ratings $R_i = 0$, then R^s has no elements ($R^s = \langle\rangle$) and $l = 0$.)

The different aspects of a free and unfree trajectory lead to the partitioning of constraints into mandatory constraints and optimisation constraints and therefore to a two level rating. The first is called *mandatory rating* and the second is called *optimisation rating*. Normally the mandatory rating contains all aspects necessary to get a free trajectory, while the optimisation rating rates a trajectory according to some optimisation criteria, as soon as the trajectory fulfils all mandatory constraints. In this distinction we do not expect that the mandatory constraints only consider getting a free trajectory. It is up to the user to define which constraints are mandatory and which are not.

The above defined rating is simple in the sense that we do not distinguish between *mandatory ratings* and *optimisation ratings*. In practice, these two types of ratings need to be handled differently, namely, an optimisation rating should only be considered after all mandatory ratings are fulfilled. To this end, the ratings will be combined as follows. As long as there is at least one section with a mandatory rating that is not fulfilled, the rating consists only of mandatory ratings and all optimisation ratings are ignored. On the other hand, if all mandatory ratings at all sections are fulfilled then we only consider optimisation ratings (with a negative sign). For this case, a lower bound c_o gives the minimal rating that we want to reach. After c_o has been reached, our planner stops and reports success.

We change the properties of the rating function for some criteria as follows. Again, the value of the rating has to represent the actual condition of the constraint in a trajectory as well as possible and the rating should have as few local minima as possible. But now we assume that the constraint of a mandatory rating is better fulfilled if the rating for the trajectory delivers *lower* values, whereas an optimisation rating must deliver *higher* values for the constraint to be better fulfilled. Each rating has to be equal or greater than zero. For the mandatory ratings we assume that the value zero represents the condition when the constraint is fulfilled.

The overall rating R_i of a trajectory section i is a combination of all mandatory ratings $r_{i,j}, 0 \leq j \leq l$ and optimisation ratings $r_{i,j}, l < j \leq n$ for this trajectory section. Using the simple rating we would not distinguish between *mandatory ratings* and *optimisation ratings* apart from the sign. But as we have mentioned before, these two types of ratings have to be handled differently, that is, an optimisation rating should only be considered after all mandatory ratings are fulfilled. For this we consider all mandatory ratings $r_{i,j}, 0 \leq j \leq l$ if for any section the mandatory constraints are not fulfilled. If all mandatory ratings are equal to zero, then only the optimisation ratings $r_{i,j}, l < j \leq m$ are considered. Furthermore, a lower bound c_o gives the minimal considered rating.

Definition 8.3 Extended Rating
Let T be a trajectory with $m+1$ base points. For all mandatory ratings $r_j, 0 \leq j \leq l$ and optimisation ratings $r_j, l < j \leq n$ let $f_j \geq 0$ and $e_j \geq 0$. Furthermore, let a bound c_o be given. The extended rating R_i^e of a section i is given by

$$R_i^e = \begin{cases} \sum_{j=0}^{l} f_j r_{i,j}^{e_j} & : \quad \text{if } \exists i\colon 0 < \sum_{j=0}^{l} f_j r_{i,j}^{e_j} \\ -\sum_{j=l+1}^{n} f_j r_{i,j}^{e_j} & : \quad \text{if } \forall i\colon 0 = \sum_{j=0}^{l} f_j r_{i,j}^{e_j} \\ c_o & : \quad \text{if } R_i^e < c_o \end{cases}$$

The overall rating R^e for the whole trajectory is the sorted sequence of all trajectory section ratings $R_i^e \neq c_o$. Let $s(i)$ be a bijective function with a range and image of $1 \leq i \leq m$.

$$R^e = \langle R_{s(1)}^e, \ldots, R_{s(l)}^e \rangle \quad , \quad \begin{array}{l} \text{with } R_{s(i)}^e \geq R_{s(j)}^e \text{ if } i < j \\ \text{and } \forall i \leq l : R_{s(i)}^e \neq c_o \text{ and } \forall i > l : R_{s(i)}^e = c_o \end{array}$$

(If all ratings $R_i^e = c_o$, then R^e has no elements ($R^e = \langle\rangle$) and $l = 0$.)

As a consequence, the optimisation rating is only considered if all mandatory ratings have reached zero. If c_o is chosen to be greater than or equal to zero, then the optimisation ratings will not be considered at all in the planning process since the planner stops as soon as all mandatory ratings signal fulfilment.

Let us next focus on how to compare two trajectories with their ratings and how to define an order on trajectory ratings. There are two requirements for such a relation. Firstly, we would prefer it if one section of one trajectory T_1 is worse than all sections of the other trajectory T_2, then T_1 should be considered worse then T_2 independently of the remaining section ratings of T_1. In other words, if there is a particularly bad section in one trajectory then we do not care how good or bad the other sections are. Second, in order to measure improvement we would like to introduce some kind of ε value by which the bad sections of the two trajectories in question must differ before we put them in relation. The following "epsilon smaller" relation fulfils both conditions.

Definition 8.4 Epsilon Smaller Relation
Let T_1 and T_2 be two trajectories. The trajectory T_1 consists of $n'+1$ base points and has the extended rating $R_1^e = \langle R_{1,1}, \ldots, R_{1,n} \rangle$ ($n \leq n'$), and the trajectory T_2 consists of $m'+1$ base points and has the extended rating $R_2^e = \langle R_{2,1}, \ldots, R_{2,m} \rangle$ ($m \leq m'$). (Let $n = 0$ if $R_1^e = \langle\rangle$ and let $m = 0$ if $R_2^e = \langle\rangle$.)

Let k, with $1 \leq k \leq m$, be the index such that the following condition holds

$$(\forall i, 1 \leq i \leq k : R_{2,1} = R_{2,i}) \wedge (\forall i, k < i \leq m : R_{2,1} > R_{2,i})$$

Let l, with $1 \leq l \leq \min(n,m)$, be the index such that the following condition holds

$$(\forall i, 1 \leq i \leq l : R_{1,i} \leq R_{2,i}) \wedge (\forall i, l < i \leq \min(n,m) : R_{1,i} > R_{2,i})$$

We say the rating R_1^e is epsilon smaller than the rating R_2^e ($R_1^e <^{\varepsilon} R_2^e$ or $T_1 <^{\varepsilon} T_2$) if

$$(\forall i, 1 \leq i \leq \min(k,n) : R_{1,i} \leq R_{2,i}) \wedge (n < k \vee \exists i, 1 \leq i \leq k : R_{1,i} < R_{2,i}) \qquad (8.1)$$

and

$$(n = l \wedge n < m) \vee \left(\exists j, 1 \leq j \leq l : \sum_{i=1}^{j} (R_{2,i} - R_{1,i}) > \varepsilon j \right) \tag{8.2}$$

This relation ensures the following. First, one of the largest ratings needs to get smaller (see Equation 8.1) and second, the sum of the other ratings needs to decrease at least by an epsilon value (see Equation 8.2). If epsilon is equal to zero, then only one of the highest ratings has to decrease. If epsilon is larger than zero, then it is not only up to one highest rating to decrease but it is also necessary to reduce other bad ratings by epsilon with respect to their position in the sorted sequence. Note that the value by which a section rating has to improve is weighted by the index of its rating in the rating sequence. Hence, if the improvement of the rating does not take place at the worst section, then the improvement must be higher.

With this relation we hope to select trajectories that improve the situation in a certain way. On the one hand, we would like to stay focused on a section with the worst rating. On the other hand, we would also like to support improvements that affect bad sections in the vicinity of the selected worst section (which is the case if we do only local changes).

We would like to mention, that in practice, the epsilon smaller relation need not be tested on a whole trajectory but only on selected sets of neighbouring sections. If we only select one worst section for manipulation, it is sufficient to set the focus for the epsilon smaller relation on all neighbouring sections that might get changed by a local change of the selected worst section. The number of affected sections depends on the trajectory type but is usually much smaller than the overall number of sections.

After we have defined a relation on the sorted sequence of section ratings, we need to define under which condition the sorted sequence is "zero" depending on a given lower bound c_o.

Definition 8.5 Zero Relation
Let T be a trajectory consisting of $n'+1$ base points with extended rating $R^e = \langle R_1, \ldots, R_n \rangle$ ($n \leq n'$). We say the rating R^e is zero for c_o ($R^e =^{c_o} 0$ or $T =^{c_o} 0$) if R^e is the empty sequence, that is, $R^e = \langle\rangle$ ($n = 0$).

For a given start rating R^e_s, an epsilon value greater zero, and a finite sequence of ratings $R^e_s >^{\varepsilon} R^e_1 >^{\varepsilon} \cdots >^{\varepsilon} R^e_n$ zero is eventually reached if we assume that a trajectory can only have a limited number of base points.

In the description of our algorithm we assume that ε and c_o are global parameters and we do not need to add them to the parameter list of each function. The condition for value ε is that it is greater than 0.

The function *rating* `rate` (*trajectory T*) returns the extended rating of the trajectory T (see Definition 8.3). We have given two examples how single ratings may be calculated. First, the torque and force rating on page 73 (`rateTorqueAndForce`), second the collision rating on page 93 (`rateCollision`). The optimisation rating is considered if the sum of the mandatory ratings is equal to 0. Note that recalculation of a rating is only necessary for parts of the trajectory which have changed in a planning step. This certainly depends

on the trajectory type. (In the worst case, the whole trajectory changes, if one base point is changed.)

Furthermore, let *bool* `zero` (*trajectory* T) determine if the rating of T is smaller than a lower bound c_o. Since an optimisation rating can be unbounded it is necessary to have a lower bound, where we can stop the planning process.

The function *bool* `smaller` (*trajectory* T_1, *trajectory* T_2) compares two trajectories and decides, if trajectory T_1 is epsilon smaller than T_2 (see Definition 8.4). As we have mentioned above, we only want to apply the epsilon smaller relation to those trajectory sections that change when the selected worst section is modified by our planner. For example, if a base point changes its position in a polynomial point-to-point motion, then only the previous section and the next section need to be considered in the epsilon smaller relation. In a bang-bang path motion, the two previous sections and the two following sections have to be taken into account. In the following description we assume the worst case, that is, all sections of the current trajectory change.

bool `smaller` (*trajectory* T_1, *trajectory* T_2)

```
/* calculate ratings */
rating R1 = rate (T1)
rating R2 = rate (T2)

/* if rating T2 is already zero, than T1 cannot be smaller */
if (zero (T2))
    return false

/* if rating T1 is epsilon smaller than T2 or T1 is zero */
if (R1 <ε R2 ∨ zero (T1))
    return true
return false
```

The epsilon smaller relation is vital to our planning process. It lets our planner decide when to accept changes to a trajectory and when to discard them. The epsilon value itself is a tuning parameter to our algorithm. If ε is too large then our planner will only accept changes that are “big steps”. If ε is too small then even the slightest improvement may distract our planner from more successful directions.

8.2 Finding Alternative Trajectories

We want to analyse different operations for manipulating a given trajectory. We change the trajectory by moving, adding, or deleting base points and sections respectively. The

aim is to improve the rating for the resulting trajectory such that the new trajectory can be used for further modification. If none of the single operations result in a better rated trajectory, then our planner performs a random movement.

We use the description of a trajectory from Chapter 5 (Trajectory Generation), where the trajectory is given by its base points $\vec{b}_0, \ldots, \vec{b}_m$. The section S_i is the trajectory between t_{i-1} and t_i. In the following, let $\vec{b}_i$ be a time configuration of dimension n consisting of the configuration $\vec{\omega}_i$ and the time t_i, without consideration of the epsilon and delta values.

8.2.1 Moving Sections

The section S_i is moved by moving the base points $\vec{b}_{i-1}$ and $\vec{b}_i$. We move both base points since the rating is given via sections. If only one base point were moved then it would not be possible to determine whether the previous or the following base point should be preferred. Apart from that, by moving both base points the planner has more control on the run of the resulting trajectory. A base point can be moved in all directions within the configuration space and also in time. The movement in the dimensions of the configuration space is usually limited by the joint limits, but it could be unlimited as well. The movement in the dimension of time is limited by the time of the previous and the following base point. The problem of moving base points is, that with an increased number of dimensions in the configuration space, the number of directions in which the base point can be moved increases exponentially. Therefore, it is necessary to have a reasonable strategy for moving the base points.

Note that the first and the last base point cannot be moved, neither in position nor in time since they represent the start and goal configurations. The position $\vec{\omega}_i$ and time t_i of the other base points $\vec{b}_i$ may be moved by vectors $\vec{v}_{i,j}, j = 0, \ldots, n-1$ in a positive or a negative direction. It is not necessary to move both base points of the selection section. The vectors $\vec{v}_{i,j}$ are the unit vectors of the configuration space dimensions. The movement is only possible if the resulting position is inside the configuration space limits and the new time of $\vec{b}_i$ has a delta distance $\delta > 0$ to the previous and the following base point $t_{i-1} + \delta \leq t_i \leq t_{i+1} - \delta$.

Let us turn to the movement of the robot in the physical world. When a dimension of the configuration space is changed, then the resulting movement of the real robot is dependent on the depth of the affected joints in the robot's topology (see Figure 4.2) and the dependency functions of the robot (see Definition 4.6). If, for example, a joint nearer towards the robot's base is moved, a lot more joints change their position than during a movement in a joint further away from the base. Apart from that, the distance the top joints travel normally increases if a joint nearer to the robot's base is moved. The dependency function of the joint influences the amount of changing as well, for example, the real movement could be a scaled version of the configuration space movement. If there are no dependency functions (i.e. there is a one-to-one mapping between robot joints and configuration space dimensions) then the spectrum of possible movements can be quite large. The same amount of change in some configuration space dimension may be responsible for rotating the outermost link by 90 degrees (a finger, for example) or may be responsible for rotating the whole robot by 90 degrees.

A parameter the user may choose is the order in which the joints of the base points

are tested. One reasonable order would be to test joints in the order they appear in the robot's topology tree, either bottom up ("upwards") or top down ("downwards"). (If we have dependant joints, then the first joint of the dependant joints gives the position in the bottom-up or top-down sequence.)

If a section S_i is moved, then in the worst case there are $(3^n)^2 - 1 = O(2^{nO(1)})$ possibilities to readjust the base points $\vec{b}_{i-1}$ and $\vec{b}_i$, if we take all pairs of possible changes. The first base point could be changed arbitrarily in all joints (3^n), the same holds for the second base point (3^n), and at least something has to change (-1). If only one dimension is allowed to change for each base point, then the number of possibilities is reduced to $(2n^2) + 4n = O(n^2)$. This number results from the number of possibilities to change a dimension of the first base point ($2n$) combined with the possibilities of one dimension of the second base point ($2n$), plus the number, if only the first or the last base point is changed ($2(2n)$). The complexity can further be reduced if we always change the same dimension in both base points of the current section. This results in $(3^2 - 1)n = O(n)$ possibilities per section move. In our algorithm we use the latter method.

Let *bool* `move` (*trajectory* T_n, *trajectory* T_o, *int i*, *int j*, *double* d_p, *double* d_n) be a function that moves section i. The dimension of the time configuration that is moved is j and the trajectory T_o has m base points. The parameters d_p and d_n give the amount by which the previous base point and the next base point should be moved. Here, previous and next refer to the current selected section i, that is, the previous base point of section i is $\vec{b}_{i-1}$ and the next base point of section i is $\vec{b}_i$. The minimal delta time δ between two consecutive base points is a global parameter. If the section can be moved, then `true` is returned, otherwise `false`. The modified trajectory is written to T_n.

---

bool `move` (*trajectory* T_n, *trajectory* T_o, *int i*, *int j*, *double* d_p, *double* d_n)
pre-condition: $0 \leq j < n, 0 < i \leq m, \delta > 0$

$T_n = T_o$
/ check if the first or last base point is moved */*
if $((i-1 == 0 \ \vee d_p \neq 0) \ \wedge (i == m \ \vee d_n \neq 0))$
    return `false`

/ get and change the base points */*
basepoint $b_p = T_o(t_{i-1})$
basepoint $b_n = T_o(t_i)$
$b_{p,j} = b_{p,j} + d_p$
$b_{n,j} = b_{n,j} + d_n$

/ distinguish between joint and time configuration and check limits */*
if $(j < n-1)$ {
    if $((b_{p,j} < \omega_j^{\min} \ \vee b_{p,j} > \omega_j^{\max}) \ \vee (b_{n,j} < \omega_j^{\min} \ \vee b_{n,j} > \omega_j^{\max}))$
        return `false`

```
}
else {
    if (b_{p,j} > b_{n,j} − δ)
        return false
    if (d_p ≠ 0 ∧ b_{p,j} < t_{i−2} + δ)
        return false
    if (d_n ≠ 0 ∧ b_{n,j} > t_{i+1} − δ)
        return false
}

/* make new trajectory */
T_n(t_{i−1}) = b_p
T_n(t_i) = b_n
return true
```

In contrast to this, Glavina [Glavina, 1991] suggests movements orthogonal to the actual moving direction in configuration space to improve the situation. This results in $2n-2$ possibilities if the dimension of the configuration space is n. Using an orthonormal movement concerning the movement direction in configuration space would result in calculating first the base vector of the orthonormal base $\vec{v}_{i,0} = (\vec{b}_{i+1} - \vec{b}_{i-1})/|\vec{b}_{i+1} - \vec{b}_{i-1}|$. From this vector an orthonormal base using $\vec{v}_{i,0}$ as first base vector can be generated. The disadvantage of this method is, that for different moving directions, the amount of the movements of the real robot could be very different. Two differently generated, orthonormal bases from different moving directions may lead to a wide spectrum of real robot movements in one case or, in the other case they may contain a lot of very similar positions in physical space.

In Figure 8.1, an example of a base point movement is depicted. In this example, no obstacle is moving and we are only interested in the left finger of the robot, that is, we allow only joint J_2 and J_3 to move. The two upper pictures represent the configuration space if only ω_2 and ω_3 are moved. On the left side, one can see the movement for a polynomial point-to-point motion and on the right side the movement for a bang-bang path motion is shown. The red lines represent the start situation consisting of a trajectory with three base points. The yellow path demonstrates the moving of the middle base point along the dimensions of the configuration space. Every step leads to a better exact trajectory. Finally, this movement results in the new green trajectory. In the right case the trajectory is already free. It can be seen, that in the path motion the exact trajectory does not touch the middle base point.

The example shows that the step size λ of the movements in the configuration space is an important parameter. If the step size λ is too short, then the change in the rating may not be significant enough, since the new rating has to be epsilon better than the old one. On the other hand, steps that are too big may not follow the gradient of the rating correctly. This parameter has to be chosen by the user. In the upcoming chapter with experimental results, we will demonstrate the effect of various step sizes on the planned trajectories and the planning time. If a step in one direction was successful, then the step

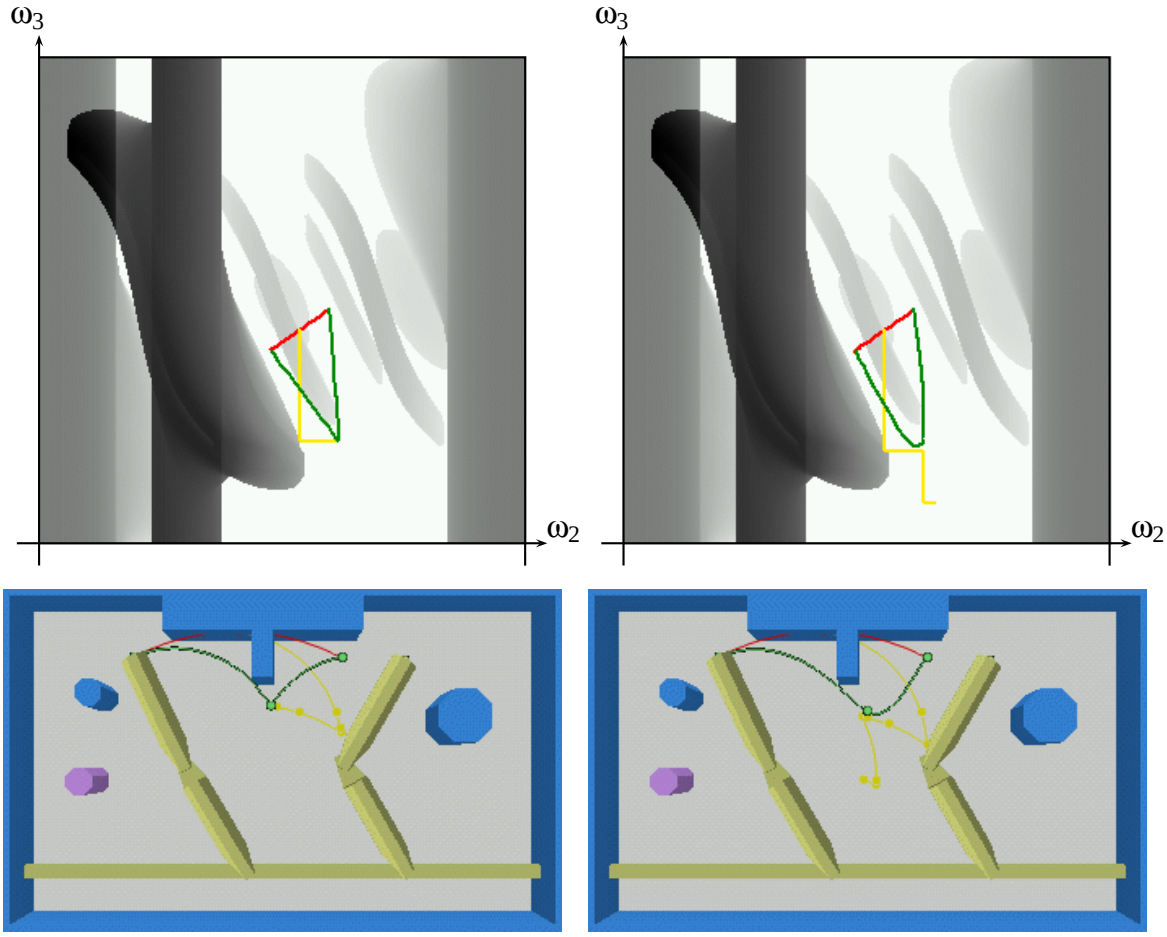

Figure 8.1: *Movement of a base point. Left half shows a polynomial point-to-point motion. Right half shows a modified bang-bang path motion*

size is doubled and the same direction is tested again, until no more improvement can be achieved.

In contrast to this, Baginski [Baginski, 1999] moves a base point such that the colliding joint of the robot does an orthogonal movement in the 3D object space. This is a very good strategy for getting a robot out of an obstacle, but it has its limitations if other constraints apart from collision need to be considered too. For example, if the trajectory is unfree because the dynamic limit of one joint is exceeded, then it is quite difficult to decide which joint has to be moved. Since we want to give an algorithm for arbitrary ratings we cannot use such a strategy. Another reason why we need a more general strategy is that our base points may not be points on the exact trajectory.

The new rating of the trajectory has to be better than the old one. More precisely, the rating R_n of the new trajectory T_n has to be epsilon smaller than the rating R_o of the old trajectory T_o, for some epsilon that is given. Not all sections of the new trajectory have to be recalculated, only the sections which change in the vicinity of section S_i, and that depends on the type of the exact trajectory. The epsilon smaller relation is only checked

for the changed trajectory sections around S_i. We have discussed the influence of adding or deleting base points on exact trajectories in Chapter 5.

Let *bool* `moveSection` (*trajectory* T_n, *trajectory* T_o, *int* i) be the function that tries to change the section i of trajectory T_o. The global parameter λ determines the distance a base point is moved in one dimension of the configuration space. If a new trajectory can be found that is epsilon smaller than the old one, then the trajectory T_n is written and `true` is returned, otherwise `false` is returned. We have implemented two variants of this function. The first one always takes the first improving modification that can be found while the second one takes the best modification. The effect of each variant will be shown in the chapter with experimental results. If the global parameter `first` is set to `true` then the first modification is taken, otherwise we take the best modification.

For function `moveSection` we need a global list l that holds as tuples all possible moving directions for each dimension of the configuration space in some order. Each element of l is a tuple (j, d_p, d_n), where j determines the dimension of the configuration space ($j \in \{0..n\}$), d_p is the movement direction for the previous base point ($d_p \in \{-1, 0, 1\}$), and d_n gives the moving direction for the next base point ($d_n \in \{-1, 0, 1\}$). It holds that either $d_p \neq 0$ or $d_n \neq 0$. This list will be generated beforehand and contains 8 ($= 3 \cdot 3 - 1$) tuples for each dimension and hence $8n$ tuples in total. During movements, the list will keep its length but the order of the elements in the list will be changed. The list defines the order in which we check the dimensions of the configuration space. One of our heuristics is to place the tuples of a dimension that has been successfully used to the front of the list since it is likely that the same dimension will be successful in subsequent movements.

The list operation *listElement* `listFirst` (*list l*) returns the first element of the list. If the list is empty then `null` is returned. Operation *listElement* `listNext` (*listElement e*) returns the element after e in a list. If there is no element following e then `null` is returned. Operation *list* `listRemove` (*list l*, *int i*) removes all elements for dimension i from the list and returns the new list. *list* `listInsert` (*list l*, *int i*) adds all moving possibilities of dimension i to the front of list l. Hence, `listRemove` removes 8 tuples from the list while `listInsert` inserts 8 tuples. The content of a list element can be read with *int* `listReadTupel` (*listElement e, int i*), where $i \in \{0, 1, 2\}$ since each tuple consists of three elements.

---

bool `moveSection` (*trajectory* T_n, *trajectory* T_o, *int* i)
pre-condition: $0 < i \leq m$, `!zero` (T_o)

---

listElement $e =$ `null`
bool `movedAny` = `false`, `movedOne` = `false`, `moved` = `false`
trajectory $T_t = T_o$, $T_s = T_o$, $T_a = T_o$
$T_n = T_o$

/ as long as the rating gets better try to move the section */*
`movedOne` = `true`

```
while (movedOne ∧ !zero (T_n)) {
    e = listFirst (l)
    movedOne = false

    /* for all elements in the list */
    while (e ≠null ∧ !zero (T_n)) {
        int j = listReadTupel (e,0)
        double d_p = λ* listReadTupel (e,1)
        double d_n = λ* listReadTupel (e,2)

        /* if movement is possible, double distance and try again */
        moved = true
        while (moved ∧ !zero (T_s)) {
            moved = false
            if (move (T_t,T_a,i,j,d_p,d_n)) {
                if (smaller (T_t,T_s) ∨ zero (T_t)) {
                    moved = movedOne = movedAny = true
                    T_s = T_t
                    d_p = 2.0 * d_p
                    d_n = 2.0 * d_n
                }
            }
        }

        /* take next element in list */
        e = listNext (e)

        /* if mode is "move first", reorganise list and start from the beginning */
        if (first ∧ moved) {
            T_n = T_a = T_s
            l = listRemove (l, j)
            l = listInsert (l, j)
            e = null
        }
        /* if mode is "move best", store best */
        if (!first) {
            if (smaller (T_s, T_n))
                T_n = T_s
            if (e == null)
                T_a = T_n
        }
    }
return movedAny
```

If `first` is `true`, this algorithm takes the first successful modification according to the order of the list and then modifies the list. Hence, movements in a dimension which have been successful are moved to the head of the list. If `first` is `false`, then the best modification is taken. In the experimental result chapter we discuss these two different strategies in more detail. We would expect the first strategy to be faster and the second strategy to find more solutions but probably to take more time.

The function `moveSection` only modifies one section of the list. For the planning process it is useful to modify a complete neighbourhood of a bad section after the section has been moved. Function *bool* `moveInterval`(*trajectory* T_n, *trajectory* T_o, *int* i, *int* j) tries to move the sections S_i to S_j starting with the worst section and then modifies the neighbourhood of the worst section. In this function, the list of possible movements is generated and kept global for each single movement. Hence, successful dimensions move to the head of the list during modification of the whole interval (this applies only if `first` is `true`).

As we have mentioned before, another choice we have to make is, whether the possible movements start with the joints near to the base or with a joint at the other end of the robot's topology. To improve a collision it seems to be a good strategy to start at the bottom of the robot, as upper joints, which are in full collision, would not be able to improve the rating. But if we consider other ratings as well, this rationale no longer applies. In our experiments we have tested both variants. Let the global parameter `down` determine the search direction and the generation of the global defined list l respectively. If `down` is `true` the search starts with the top joints, otherwise if `down` is `false` the joints nearer to the base are tested first. Furthermore, we also have tried to use a randomisation during this list generation. If `random` is `true` we start at a random position in the joint hierarchy, but then we again move bottom-up or top-down through the robot's hierarchy. Let *list* `generateList` () be the function generating this list depending on the global parameters `down` and `random`.

The function *int* `maxRating` (T) returns the section index with the worst rating. If there are more sections with the same maximal rating, then the smaller index is returned. If all ratings are zero, then -1 is returned.

---

bool `moveInterval` (*trajectory* T_n, *trajectory* T_o, *int* i, *int* j)
pre-condition: $0 < i, j \leq m$, $i \leq j$, `!zero` (T_o)

---

int s = `maxRating` (T_o)
bool `moved` = `true`
bool `movedOne` = `true`
bool `movedAny` = `false`

/* *generate the global list* l *for the movements depending on* `down` *and* `random` */
list l = `generateList`()

```
/* move as long as the section can be improved */
while (movedOne ∧ s ≥ i ∧ s ≤ j) {
    movedOne = false

    /* move the worst section */
    moved = movedOne = moveSection(T_n,T_o,s)

    /* move the previous sections */
    for (s = i − 1; s ≥ i ∧ moved; s − −)
        moved = moveSection(T_n,T_o,s)

    /* move the next sections */
    moved = movedOne
    for (s = i + 1; s ≤ j ∧ moved; s + +)
        moved = moveSection(T_n,T_o,s)

    /* set moved any and get new max rating */
    if (movedOne)
        movedAny = true
    s = maxRating (T_n)
}

return movedAny
```

Finally, we want to summarise the parameters of the moving process. First of all, there is the *minimal time distance* δ two consecutive base points may have, which could influence the possibility of a move in the time dimension. The *step size* λ determines the amount a base point is moved while searching for better rated trajectories. Then there are four possibilities the dimensions of the configuration space are used to modify the trajectory. *Bottom-up* (`down = false`) indicates that search directions is bottom-up in the hierarchy of the robot's joints, otherwise it is *top-down*. The third type is to take the *best* choice (`first = false`), which searches all dimensions and takes the best trajectory. The last parameter in this category is, if the starting dimension in the bottom-up and top-down search is chosen *random* (`random = true`). Finally, ε determines the epsilon smaller relation and c_o the zero relation, which indicate if a trajectory is better than another or even has 'zero rating'.

8.2.2 Adding Sections

A new base point is added in a section S_i by choosing a new base point $\vec{b}_j$ with $t_{i-1} + \delta \leq t_j \leq t_i - \delta$ and inserting it into the old trajectory. If a section is too short (in terms of time), that is, the delta distance cannot be satisfied, then we cannot add a base point to the selected section. (Recall that the minimal time distance *delta* is a global parameter of our algorithm.) If the delta condition can be satisfied, then the time of the new base point is

chosen in the middle of the old section S_i: $t_j = t_{i-1} + \frac{t_i - t_{i-1}}{2}$. The values of the new base point are taken from the current trajectory at time t_j, as we hope to minimise the change to the trajectory rating.

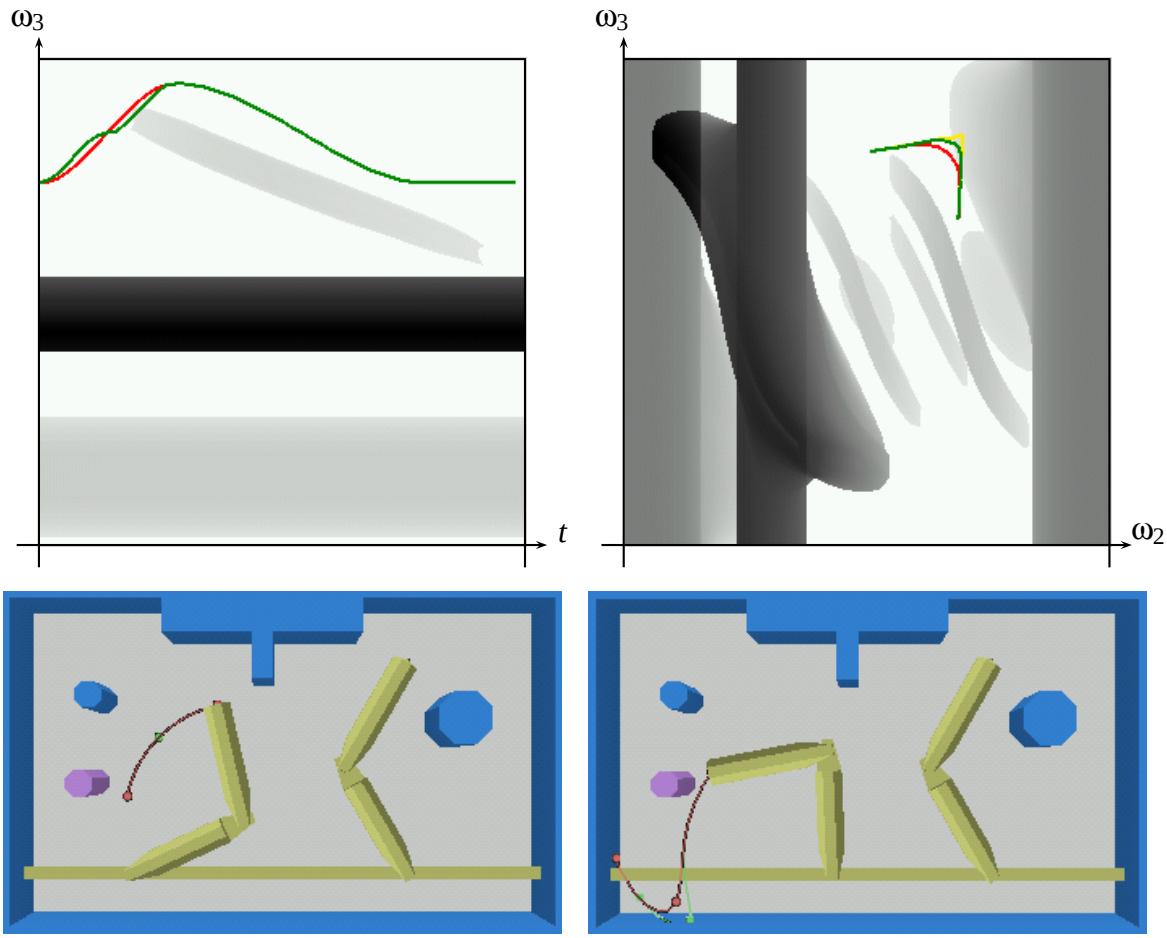

Figure 8.2: *Adding a base point can worsen a trajectory. Left half shows a polynomial point-to-point motion. Right half shows a modified bang-bang path motion*

If the trajectory consists only of two base points (start and goal configuration), then two base points are added. This is reasonable because after that the planning process can immediately continue by moving a whole section.

Sometimes the rating of the new trajectory may be worse than the old one, since the trajectory may change its run simply due to the existence of a new base point (even if the new base point is on the old trajectory). Clearly, this depends on the trajectory type that we use. This effect is particularly apparent for point-to-point motion types since the robot comes to a full stop at the new base point, but it also happens with path motions.

In Figure 8.2, two examples can be seen that illustrate this behaviour. The left example uses a polynomial point-to-point motion, while the right example shows a modified bang-bang path motion. In both cases, the insertion of a new base point into an existing free trajectory leads to an unfree trajectory. On the left side, all joints other than J_3 are kept

still and the time varying obstacle is moving. The red line shows the polynomial point-to-point trajectory over the time when there are only three base points. The green line, which is in collision, has one additional base point. The geometric path is identical in both trajectories (see Figure 8.2, lower left image) but since the robot comes to a full stop at the new base point, it collides with the moving obstacle shortly after the new stop (see configuration space in Figure 8.2, upper left image). On the right side we have the same situation as in the example for moving a base point. Only joints J_2 and J_3 can move and the time varying obstacle does not change its position. Inserting a base point on the red trajectory changes the run from the red to the green trajectory. The resulting trajectory lies partly inside the obstacle region. In this example, a change in the geometric path is the reason for the collision.

To compensate for this behaviour, every time a base point is inserted, we subsequently move the two new sections and the surrounding sections (identical to the move operation in the previous section) to find a trajectory that is better than the old one. Depending on the type of exact trajectory, the neighbours have to be considered as well, since the run of the trajectory may change in these sections too (see Chapter 5). For the addition of a new base point in combination with a subsequent movement of the surrounding sections, the same conditions apply as for the movement of sections alone. The new trajectory must have an epsilon smaller rating than the old one, otherwise the insertion is rejected and undone. For ease of presentation, we assume in the following that the whole trajectory is affected by an insertion.

With the delta condition we make sure, that for a given start and end time of the planning problem, we get an upper bound on the number of base points contained in a trajectory. Let *bool* `addSection` (*trajectory* T_n, *trajectory* T_o, *int* i) be the function that tries to add a new base point in the trajectory at section i. If the trajectory consists of two elements only, then two base points are added. If the addition is successful and the new trajectory is epsilon smaller, then the trajectory is modified and `true` is returned, otherwise the function returns `false`.

Let *bool* `addBasePoint` (*trajectory* T_n, *trajectory* T_o, *basePoint* b) be a function which inserts the base point b in T_o. The new trajectory is written to T_n. If the base point has been inserted, then `true` is returned. If the base point cannot be inserted because of the time limits, then `false` is returned. Let `noSection` (*trajectory* T) return the number of sections in T (which is the number of base points minus one).

---

bool `addSection` (*trajectory* T_n, *trajectory* T_o, *int* i)
pre-condition: $0 < i \leq m$, `!zero` (T_o)

---

trajectory $T_t = T_o$, $T_s = T_o$
$T_n = T_o$

/ generate new base point depending on the actual number of base points */*
if (`noSection` (T_o) > 1) {

```
    basePoint b = T_o((t_i + t_{i-1})/2)

    if (!addBasePoint (T_t,T_o,b))
        return false
}
else {
    basePoint b_1 = T_o((t_i + 2t_{i-1})/3)
    basePoint b_2 = T_o((2t_i + t_{i-1})/3)

    if (!addBasePoint (T_s,T_o,b_1))
        return false
    if (!addBasePoint (T_t,T_s,b_2))
        return false
}

/* try to move changed surrounding */
moveInterval (T_s,T_t,1,noSection(T_t))

/* check if the new trajectory is better than the old one */
if (smaller (T_s, T_o)) ∨ zero (T_s)) {
    T_n = T_s
    return true
}
return false
```

For this function, the same parameters as for the moving in the previous section have to be considered since `moveInterval` is part of the function. The only parameter that we would like to point out is the minimal time distance δ between two consecutive base points. The adding of a base point fails immediately if there is not "enough time" to insert a new base point.

8.2.3 Deleting Sections

The section S_i is deleted by dropping the base point $\vec{b}_i$. It is obvious that the resulting trajectory may be worse than the old one. Here again the operation is followed by moving the new section and the surrounding sections, as these sections could change depending on the chosen trajectory type. For ease of presentation we again assume, that the worst case happens and the whole trajectory needs to be changed.

This operation cannot be used if the trajectory consists of only one section. Let *bool* `deleteSection` (*trajectory* T_n, *trajectory* T_o, *int* i) be the function, which tries to delete the base point $\vec{b}_i$ in the base point trajectory. If deletion is successful and the new trajectory is epsilon smaller, then the trajectory is modified and `true` is returned, otherwise the function returns `false`. This function only fails if only the start and goal configurations are left.

Let *bool* `deleteBasePoint` (*trajectory* T_n, *trajectory* T_o, *int* i) be the function which deletes the i-th base point in T_o. If i is the last base point, then the last but one base point is deleted. The new trajectory is written to T_n. If the base point has been deleted, then `true` is returned. If the base point cannot be deleted because it is the first or the last base point, then `false` is returned.

bool `deleteSection` (*trajectory* T_n, *trajectory* T_o, *int* i)
pre-condition: $0 < i \leq m$, `!zero` (T_o)

```
/* delete base point */
trajectory T_t = T_o, T_s = T_o
T_n = T_o
if (!deleteBasePoint (T_t,T_o,i))
    if (!deleteBasePoint (T_t,T_o,i-1))
        return false

/* try to move changed surrounding */
moveInterval (T_s,T_t,1,noSection(T_t))

/* check if the new trajectory is better than the old one */
if (smaller (T_s,T_o)) ∨ zero (T_s)) {
    T_n = T_s
    return true
}
return false
```

No other parameters are necessary apart from those already presented in preceding sections. Deleting a base point is, as long as there are enough base points, always possible. The critical part is to reach an improved trajectory afterwards by moving the base points in the vicinity of the deleted base point.

8.2.4 Randomised Section Movement

All of the above operations are only executed if the resulting trajectory is getting an epsilon better rating. However, it is possible that the trajectory is in a local minimum and none of the above operations achieves any improvement. We need a strategy to escape such local minima.

One possibility is to define subgoals and then try to find a connection between the start and goal configurations via these subgoals [Glavina, 1991]. In such a setting our planner would be used as a local planner searching connections via subgoals, whereas

the global planner generates random subgoals and tries to find a connection via solved subgoal connections.

The disadvantage of this method is that the trajectory planned so far is lost after a new subgoal has been chosen. We suggest a strategy that still uses randomisation but is more conservative as far as the already planned trajectory is concerned. We make a local random movement with one section of the trajectory and after that we readjust the neighbouring sections.

If the trajectory consists of only one section then no randomised movement would be possible, since we cannot move the start and goal configurations. In this case we have to add a base point before making the random movement. We suggest making a randomised movement of a selected worst section S_i. This randomised movement works as follows. A random configuration for time $(t_{i-1}+t_i)/2$ is generated (satisfying joint limits) and then "applied" to each base point. All base points of the current trajectory are readjusted depending on their time relative to the time $(t_{i-1}+t_i)/2$ and the start time and the end time respectively. As a consequence, we will not lose the relative time positions of the base points.

Let $\vec{b}_r$ be the randomly generated base point. Then the dimensions representing the joints in the configuration space take a value somewhere in the limits of the configuration space. For the dimension representing the time we do no randomised movement since we do not want to change the distribution of the base points along the time axis.

In order to adjust neighbouring base points we need a function that changes the base points nearer to section i more than those which are further away. Let $\triangle t_j$ be the time distance from time $(t_{i-1}+t_i)/2$ to the time of base point $\vec{b}_j$.

$$\triangle t_j = \left| \frac{t_{i-1}+t_i}{2} - t_j \right|$$

The simplest way is to readjust all base points of the trajectory linearly depending on the amount of $\triangle t_j$. Let $\vec{\omega}_j$ be the configuration of $\vec{b}_j$ without the time. The new configuration $\vec{\omega}_j$ of the base point $\vec{b}_j$ is calculated as follows:

$$\vec{\omega}_j = \begin{cases} \frac{\triangle t_j}{\triangle t_0}\vec{\omega}_j + \left(1 - \frac{\triangle t_j}{\triangle t_0}\right)\vec{\omega}_r & : \quad j < i \\ \frac{\triangle t_j}{\triangle t_m}\vec{\omega}_j + \left(1 - \frac{\triangle t_j}{\triangle t_m}\right)\vec{\omega}_r & : \quad j \geq i \end{cases}$$

A randomised movement normally makes the rating of the trajectory worse, but delivers a new starting point for the planning process. To eliminate the risk of taking a trajectory that is too bad, we suggest generating a set of random trajectories and selecting the one with the best rating. Moreover, it is reasonable to generate new trajectories as long as the new one is better than the last one.

In Figure 8.3, five randomly generated trajectories can be seen. The red trajectory (the one on the far left) is the original trajectory while all others are generated randomly. The highest rated section of the red trajectory is near the starting position of the left finger of the robot. For each of the five generated trajectories, the best out of ten random trajectories was taken. Note that the base points further away from the first section are much closer to their original position than the earlier base points.

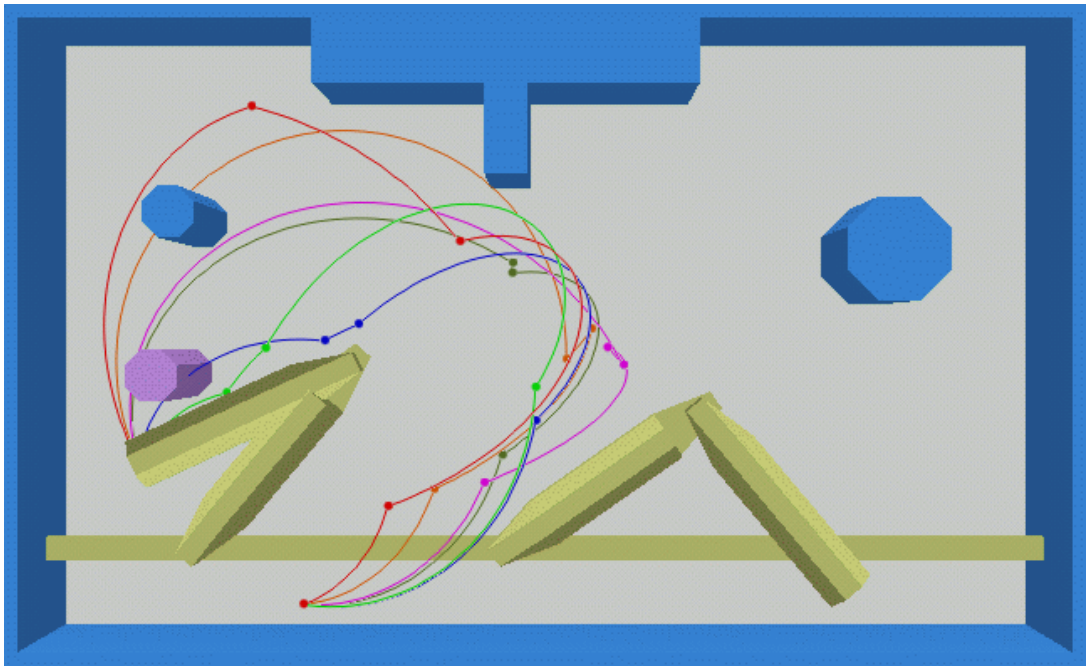

Figure 8.3: *Set of five random trajectories. The red trajectory on the far left is the original one. Each random trajectory has been taken out of ten randomly generated trajectories.*

In the rest of this chapter, let *void* `randomSection` (*trajectory* T_n, *trajectory* T_o, *int* i) be the function that creates a new trajectory T_n based on the given T_o by randomly modifying section S_i. A global parameter of this function is the number of *maximal random tests* before choosing the best one.

8.3 Trajectory Planning Algorithm

After we have described single base point operations in the last section we can now present the overall planning algorithm. We first want to describe the algorithm and then analyse it with respect to termination.

All single base point operations are always applied to the worst section, since this is the location where an improvement can most likely be achieved. Let `max` denote the maximal number of random section movements that we allow and let `noSection` (T) return the number of sections in T.

bool `planTrajectory` (T)

/ randomised movement counter */*
int $r = 0$

```
/* do until the rating is zero, or no more randomised movements allowed */
while ((r < max) ∧ !zero(T)) {
    moved = false

    /* move the worst section as long as it gets better */
    if (moveInterval (T_t,T,1,noSection (T)))
        T = T_t

    /* if rating is not zero add a base point */
    if (!zero(T)) {
        int s = maxRating (T)
        if (!addSection (T_t,T,s)) {
            /* if no add was possible delete a base point */
            if (!deleteSection (T_t,T,s)) {
                /* if no delete was possible move randomised */
                if (r++ < max)
                    randomSection(T_t,T,s))
            }
        }
        T = T_t
    }
}
return zero(T)
```

We first try to improve the rating of the trajectory by moving the whole trajectory with `moveInterval`. The function always moves the worst section first and then those sections immediately to the left and to the right of the worst section, readjusting all of its neighbours. The index of the section may change during this process. If `moveInterval` returns and the rating is not zero then a new base point is added at the worst section and the moving starts again. If neither moving nor adding is possible the worst section is deleted. If this is also not successful, then a randomised move is executed, provided that r is smaller than `max`.

An important issue that we have not paid attention yet is the choice of the step size λ. The closer the current trajectory is to the optimal rating the harder it gets for the planner to actually find a trajectory with the optimal rating. Consider a multidimensional function that we want to minimise. As long as we are far away from the minimum any movement towards the minimum is acceptable. Once we are close to the minimum we need to reduce the size of our steps in order to make sure that we do not overshoot. In our planner we would like to have a similar behaviour. To this end, we adjust the step size λ according to the distance of the current rating from the optimum. If the rating is above some threshold, the user defined step size is taken. Once the rating drops below a threshold, the step size is reduced. We suggest a logarithmic relationship between rating and step size below the

threshold. In our experiments we use the following function.

$$\lambda' = \lambda \log_{\lambda+1}(r_{\max} - c_o + 1)$$

where λ' is the actual step size that we use in the algorithm, λ is the user defined step size, $r_{\max}$ is the maximal rating of any section, and c_o is the optimisation bound. The important property of this function is that its value is close to the user defined step size for larger ratings and only decreases significantly for ratings very close to the optimum. In our experiments we take the user defined step size as the threshold value, that is, as long as the rating is above λ we set $\lambda' = \lambda$.

Let us turn to a planning example. In Figure 8.4, the 6 degrees of freedom robot can be seen. The scene contains static obstacles and one obstacle that moves in a curve from right to left. The figure shows the progress of the planner in four states which are reached during the planning. For each of the states, the trajectory reached at that state is indicated by a sequence of images.

Each column shows one planning state (one trajectory) at different time steps. From top to bottom the time increases by five seconds with each image. From left to right the rows depict the scene for four different planning states at the same point in time. The column on the far left shows the initial trajectory to which two base points have been added. The moving obstacle collides with both fingers of the robot and one finger collides with the static obstacle in the top of the box. In the second column, the two base points have been moved in the configuration dimensions responsible for the right robot finger. In the third column, more base point dimensions have been moved and this time the left finger is already free while the right finger is still struggling for a suitable side step to avoid the moving obstacle. Finally, the column on the far right shows the free trajectory. To solve this problem, our planner had to add two base points and to move them slightly.

8.4 Summary and Analysis

In this chapter, we have presented our main algorithm for planning motions in time-varying environments. The algorithm is based on various subroutines for rating and trajectory generation that we have described in the previous chapters.

In the following, we would like to give an overview on the global parameters of the algorithm. There are parameters which influence the choice of the next improved trajectory. These parameters determine the strategy for testing the dimensions of the configuration space. Two parameters exist that determine how the moving is done for a single dimension. Two more parameters determine if one trajectory is better than another trajectory. In addition, there are parameters for summing the ratings, influencing the weighting of the different ratings, and one parameter giving the exactitude of the ratings. Finally, two parameters determine the process of random movements. Last but not least, the trajectory type is an important parameter.

In Figure 8.5, the high-level flow chart that we have introduced in Chapter 3 is depicted again. We have added the parameters of the algorithm to the relevant steps. The flow chart does not give the exact run of the algorithm, but gives a good overview on where the parameters are important and where they influence the algorithm. The red parts indicate

Figure 8.4: *A four step planning sequence for the 6 degrees of freedom robot.*

comments (dashed ovals) or parts of the chart that have changed. There are some locations where the flow chart is an abstraction from the real algorithm. First, modifying a base point does not only consist of a move, add or delete operation. Immediately after an add or delete operation, we move affected sections to improve the trajectory. In the flow chart, these move operations are implicitly contained in the "modify base points" task. Second, when we generate a random trajectory then a whole set of trajectories is generated, before we select the best one. We have added a new decision step to the flow chart where we

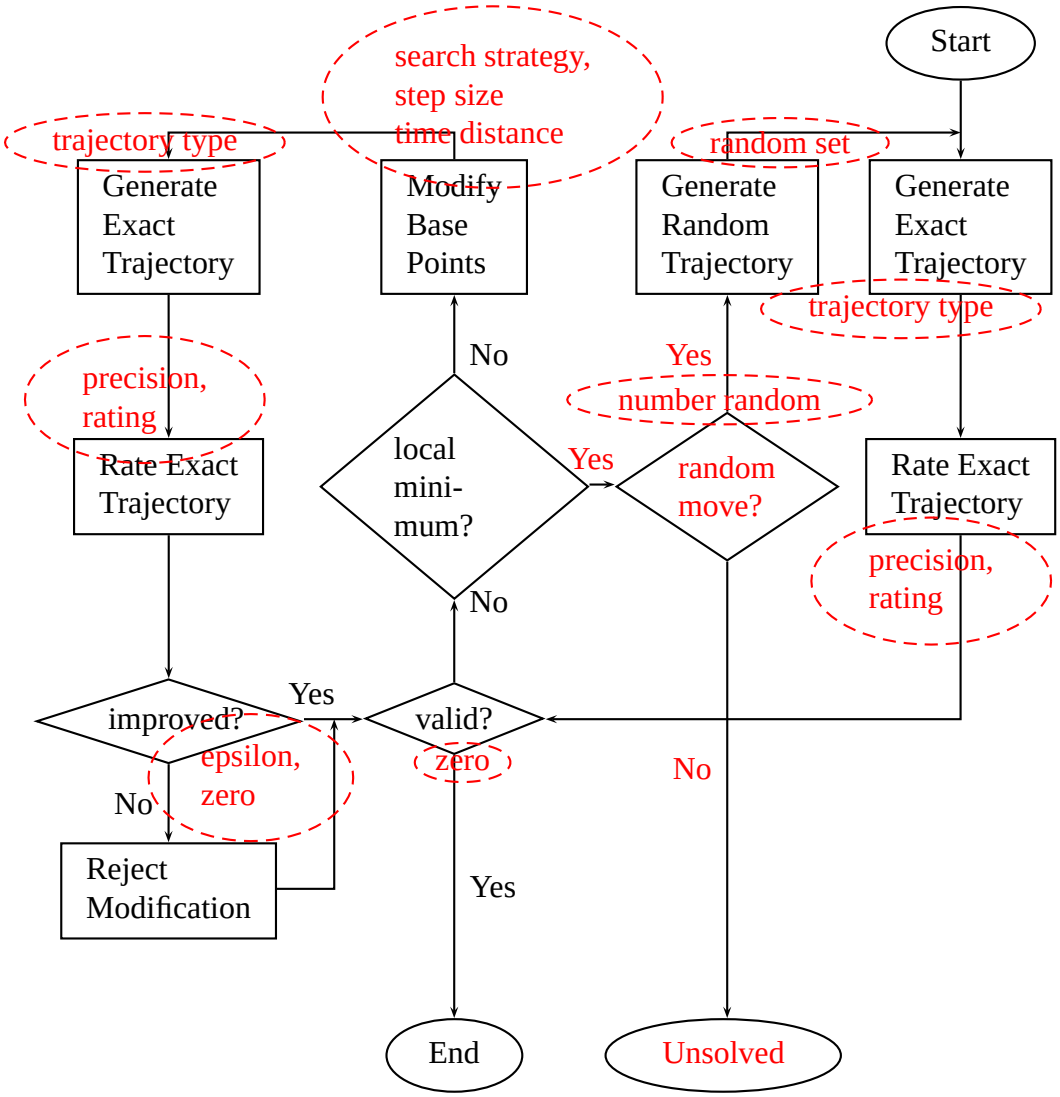

Figure 8.5: *The high-level flow chart with parameters that influence the planning process.*

check if the maximal number of random steps has been exceeded. If that happens the algorithm returns failure.

We now describe each parameter in turn. There are parameters that influence the order in which configuration dimensions are tested and modified respectively. These belong to the group of *search strategy* parameters. The Boolean parameter `down` gives the search sequence in the configuration space depending on the hierarchical structure of the robot. This can be either upwards away from the base or downwards away from the top joints. Second, the Boolean parameter `first` determines if the best trajectory of all possible

moves is taken or the first better rated trajectory is taken as the next trajectory. For the case where the first joint which improves the trajectory is taken, the joint where the sequence starts to test for an improved rating is chosen randomly if the Boolean parameter `random` is set to true.

For the modification of a single dimension in the configuration space, the *step size* λ gives the initial amount by which the base points are moved in the configuration space when searching for an improved trajectory. The *time distance* δ is important for the move and add operations. This parameter determines the minimal time distance between two base points in a trajectory.

The next parameter group influences the rating of the trajectory. The first and most important parameter is the *trajectory type*, since it influences the generation of the exact trajectory and therefore the rating. We would like to mention that the algorithm behaves quite differently for each trajectory type. For example, planning tasks exist that can be solved with one trajectory type but are unsolvable with another type. The next parameter is the *precision p*, which is important as well. Often ratings are defined via a finite number of configurations along a trajectory. The precision value determines how many discrete time steps on the trajectory are taken into account for this rating. This is done by giving a limit for the maximal distance of any point on the robot between two consecutive time steps and the maximal movement of a time-varying obstacle between time steps. The precision value is given in the unit of the modelled scene. Some ratings may ignore this value, for example, a rating maximising the time distance between the last two base points will not need this parameter. Other parameters relevant to *rating* are the multiplication coefficients f_j and exponents e_j of the ratings.

The *epsilon* value ε and the optimisation bound c_o define the epsilon smaller and the *zero* relation between trajectory ratings. These parameters determine if a trajectory has improved or is already valid.

In the case, when the basic operations of adding, moving, and deleting base points are not successful, the parameter `max` gives the total *number* of *random* movements that we allow during the planning process. The planner stops if more than this number of random movements is needed. Another parameter, called *random set* determines how many trajectories are randomly generated before the best is taken. After that, more random trajectories are generated as long as the new random trajectory is better than the last one.

In Table 8.1, all parameters or groups of parameters are collected and a short description is given for each of them.

Let us now look at the termination of the algorithm. The operations modifying a trajectory are either moving, adding, or deleting base points. All these operations decrease the rating of the trajectory by a given epsilon value. The only operation which has not to generate a better trajectory is the randomised movement. But the maximal number of randomised movements is limited.

The algorithm stops as soon as the rating has dropped below parameter c_o. Furthermore, the delta value gives the minimal distance between two base points in the time dimension. As a consequence, the number of base points per trajectory is limited. Hence, the algorithm stops after a finite amount of time, either because it has found a valid trajectory or because no more operations are possible due to the given limits regarding the number of base points, the maximal number of randomised movements, and the rating.

Parameter	Description
Trajectory type	Determines the type of the exact trajectory generated from base points.
Precision	Gives the exactitude p of the rating. A lower value gives more exact ratings.
Rating	Coefficients and exponents of the ratings, as well as the rating types.
Epsilon	Used in the epsilon smaller relation; determines if a trajectory is better than another.
Zero	Optimisation bound c_o that determines when optimisation is sufficient.
Step size	Gives the initial distance λ that a base point is moved in the configuration space.
Time distance	Two consecutive base points must have at least this time distance δ. Gives an implicit bound on the number of base points in a trajectory.
Search strategy	The order in which the dimensions of the configuration space are searched, depending on the topology of the robot. This can be bottom-up or top-down. In addition the starting dimension can be chosen randomly. Alternatively, the best dimension is taken.
Number random	Maximal number of random moves before planner signals failure.
Random set	Number of random trajectories from which the next random trajectory is selected.

Table 8.1: Parameters of the algorithm.

CHAPTER 9

Experimental Results

In this chapter, we present various experiments that we have conducted with an implementation of our planning algorithm. We first give a short introduction into the implementation and the used parameters. Thereafter we present results of the algorithm obtained in different planning scenarios.

9.1 Test Environment and Parameters

To demonstrate the usefulness of our planning approach, we have implemented our planning algorithm. The implementation is part of a robot simulation system (called *RobS*) that has been specifically designed for this purpose. Using software, we are able to simulate real world robot scenarios without the need for expensive robot hardware. Our robot simulation system can handle arbitrary robot models, environments with static and moving obstacles, planning tasks and trajectories. It allows visualisation and animation of models, configuration spaces and trajectories (all images shown in this thesis have been created using RobS).

The main implementation of our algorithm has been done in C++ while for visualisation we have used Tcl/Tk and OpenGL Mesa. All tests were run on a standard personal computer with an Intel Pentium III with 500 MHz clock speed and 128 MByte of main memory. The operating system is S.u.S.E. Linux with a 2.2.13 kernel. The version of the C++ compiler that we have used is egcs-1.1.2.

A short overview on the robot simulation system is given in the next section. Thereafter, we focus on the parameters of our planning algorithm and their possible impact on its performance.

9.1.1 RobS: a Robot Simulation System

The architecture of RobS is based on the Model-View-Control (MVC) paradigm [Buschmann et al., 1996]. The MVC architectural pattern divides an interactive application into three components. The model contains the core functionality and data. Views display

information to the user and controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.

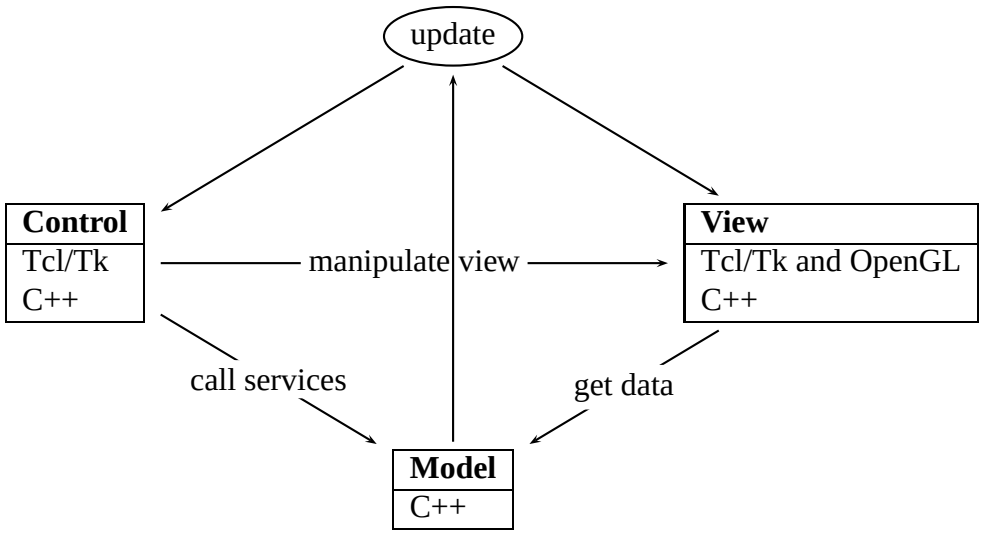

Figure 9.1: *Architecture of RobS.*

The integration of C++, OpenGL, and Tcl/Tk in the MVC architecture is illustrated in Figure 9.1. The user interface (view and control) of RobS is implemented in Tcl/Tk [Ousterhout, 1995], OpenGL [Woo et al., 1997], and C++. The free Mesa OpenGL distribution is used to generate a three dimensional view of the robot and the environment. To enable the user to input requests for the controller, a Tcl/Tk interface is used. The model part is written in C++. The models may force the update of the view and the control if it is necessary.

The scenes for our simulation system (the robot, the environment, and the planning task) can be loaded from external files. Their format is based on the Extensible Markup Language (XML). We have defined an XML document type definition (DTD) that specifies the structure of the input files. Using this DTD, it is possible to define the shape and the kinodynamic properties of the robot, the shape and trajectories of the obstacles, and the planning task itself in a structured way.

In our prototype implementation, we only support the description of robots with independent joints and a one-to-one representation in the configuration space. Moreover, we only handle the torque and force rating for forked manipulators.

9.1.2 Parameters

In the following, we turn to each parameter of our planning algorithm and discuss its possible impact on the performance.

First of all, we would like to know how the performance changes with the chosen *trajectory type*. To this end, we have implemented two representative trajectory types, the polynomial point-to-point motion and the bang-bang path motion. There are two main

differences that we expect for these two trajectory types. One concerns the calculation of the exact trajectory and the time necessary to calculate a rating. One property of the polynomial point-to-point movement is that if a base point is changed, then only the previous section and the following section will change its rating (see Chapter 5, Section 5.3.3). In contrast to this, in the modified bang-bang path motion, two sections before and two sections after the modified base point may change their rating (see Chapter 5, Section 5.4.3). As a consequence, we can expect the update of a given exact trajectory and the update of its rating after the modification of a section to be performed faster for a polynomial point-to-point motion than for a bang-bang path motion. On the other hand, we expect that with the bang-bang path motion our planner should be able to solve more problems than with the point-to-point motion. This holds in particular for time critical problems since the path motion does not stop at each base point. We expect that the modified bang-bang path motion enables faster movements using less torque and force.

Let us now turn to the *precision* parameter p of the ratings. Recall that lower values for p give higher rating precision. We expect the planning time to be reduced if a higher precision value is taken, but at the same time solutions are found less frequently. The precision gives the minimal security distance for which static collision tests are done during planning. We have to take into account that if the start or goal configuration is close to an obstacle then the chosen precision value must not be too large since otherwise the planning process may encounter difficulties in leaving the start configuration or approaching the goal configuration. The precision value is given in the unit of the physical space. For example, if the model is given in millimetre, then the unit for the precision value is in millimetres too.

Another parameter is the *step size* λ. This parameter determines how much a base point is moved initially in one dimension when we look for a better rating. We expect that changing this value will result (depending on the robot and the environment) in very different computation times. This is because too small a step size may result in many movements until the solution is found, but too large a step size may miss the situations with the best rating. The unit of step size depends on the unit represented by the respective configuration space dimension. For example, in our implementation the dimension representing rotational joints is given in degrees and the dimension representing translational joints is given in the unit of the scene (for our examples, millimetres or centimetres). The time dimension is given in seconds. We do not use any strategy to normalise the movements of the robot's joints, as is suggested for example in [Qin and Henrich, 1996]. Here the idea is that the joints nearer to the base are moved less than the joints further away to get an equal amount of movement in the physical space for each joint. In our concept, we leave this to the specification of the model. That is, we assume that in the model description, the dependency function of each joint is specified, which is responsible for scaling a configuration space movement to an appropriate movement in physical space. The only dimension that cannot be influenced in this way is the time dimension. In future it would be preferable to calculate the units of the configuration space dimensions automatically and to adjust the time scale to the planning task.

Another important parameter is the *epsilon value* which determines if a rating is epsilon smaller than another rating. Smaller epsilon values tend to find solutions even if the rating decreases very slowly, but can lead to a very slow decrease in rating even if another

dimension in the configuration space would result in a larger change of rating. In other words, the epsilon value should reflect the steepness of the rating function. If epsilon is too small, our planner will make many small steps where a few bigger steps would be appropriate. If epsilon is too large, then the planner may overlook promising search directions. It would be ideal if we could adjust the epsilon value on-the-fly during planning. For example, we could gather local and/or global information on the rating function in some intelligent way or we could analyse the history of ratings for this purpose.

Let us now turn to the parameters that determine the search strategy. The initial order in which joints are moved can either be from the base to the top joints (*bottom-up*) or the other way round (*top-down*). Additionally, we have a choice of either taking the first joint in the current order that improves the rating or we can take the joint that makes the best rating (*best*). If we have chosen the latter strategy, the order in which joints are tested is irrelevant. For the bottom-up and top-down search direction, the starting joint may be chosen randomly (*random joint*). It is quite difficult to make a prediction of the behaviour of these different selection methods. However, there should be a significant difference between taking the best joint or the first joint in the current order. Taking the best rating results in longer computation times with a faster decreasing rating, since all configuration dimensions have to be tested in each move. In contrast to this, taking the first joint that improves the rating will result in less computation time for one movement but may result in smaller overall changes of the rating. In our experiments, the bottom-up search strategy (taking the first successful joint) turned out to be a good choice in most scenarios. The reason for this may be that it is helpful to try those joints first that have a greater impact on the overall situation. Clearly, changing the robot's base joint is likely to have more impact than moving a joint in the robot's tool, e.g. a finger.

Another group of parameters control randomised movement. This group consists of a parameter that determines the maximum number of allowed random moves (*Random Number*) and the size of the set of random trajectories that we generate before we pick the best of them as the next random trajectory (*Random Set*). Most of our experiments have been solved without random moves. Our planner has used random moves only in the "trap-task" benchmark with a static environment. There, we have allowed up to 10 random moves and the set size was 25.

Finally, there is the group of rating parameters. In our experiments, we have used the rating of the collision and the torque and force rating as we have presented in Chapters 7 and 6. All factors and exponents for the rating combination have been set to 1 in all experiments.

In Table 9.1, an overview on parameters is given together with values that we have used in the experiments.

9.2 Experiments

In the following, we present various planning scenarios and test results that show the usefulness of our planning approach. We have divided our experiments into six parts. In the first part, we present a simple robot in a not-so-simple time-varying environment. In the second part, we show a typical industrial time-varying environment with a robot that

Parameter	Possible values and units
Trajectory type	polynomial point-to-point, modified bang-bang trajectory
Precision	1, 5, 10, 20, 50 (millimetre, centimetre) unit depends on unit used to model the scene
Rating	$r_c + r_t$ (not changed) $1 * \texttt{rateCollision}^{\,1} + 1 * \texttt{rateTorqueAndForce}^{\,1}$
Epsilon	10^{-8}, 10^{-3}, 1, 5, 10
Zero	0 (not changed)
Step Size	1, 5, 10, 20, 50 (degree, millimetre, centimetre, seconds) unit depends on unit of the configuration space dimensions
Time distance	0.032 sec (not changed)
Search strategy	bottom-up, top-down, best random, fixed (only with bottom-up and top-down)
Number random	0, 10
Random set	25 (not changed)

Table 9.1: Values used for parameters of the algorithm.

needs to move workpieces from a table to a moving band-conveyor. Then we take a look at benchmarks for static environments which were developed at the University of Karlsruhe. (Unfortunately, no benchmarks exist for planning in time-varying environments yet.) Then we show an example with a focus on robot dynamics. In this example we have fitted a well-known type of industrial robot, an RX90, with a very heavy hand that it must lift from the ground. In the next example, a long, slender 6-dof robot in a small room must grasp into a small sliding door. Finally, we take a look at a space robot with a high-dimensional configuration space that has to cope with asteroids.

In Table 9.2, we have collected some measurement categories in which we are interested. Our main interest during the experiments is whether a task was solved and how long it took either to solve it or to signal failure. In some experiments, we have conducted a series of tests using random start and goal configurations. In these cases, an interesting characteristic is the percentage of solved tasks. Another focus is on the percentage of the running time used for the collision test, the collision rating, and the torque rating. The number of base points in the final trajectory is another reasonable measurement category. We will not analyse all parameters and results for each scenario, rather we select those parameters that are in some way significant or representative for a given scenario.

In the rest of this chapter, we will use the following abbreviations: m for metres, mm for millimetres, cm for centimetres, min for minutes, sec for seconds, kg for kilogrammes, g for grammes, N for Newton. In addition, we use boldfaced and normal printed numbers in tables. If both kinds are used in a table, then the bold numbers indicate numbers for solved tasks while the normal printed numbers represent numbers for unsolved tasks. If we do not distinguish, then the tasks were solved for all given parameters. Red values indicate the best value for each trajectory type in a test series.

Result	Unit
Running time	seconds
Number of solved tasks	per cent
Torque rating time	per cent
Collision rating time	per cent
Collision test time	per cent
Number of base points	integer

Table 9.2: Measurement categories.

9.2.1 Two Degree of Freedom Robot

Figure 9.3 on page 129 shows an environment with a simple robot. The robot is a two degree of freedom robot which has to move in a rotating cube. The figure shows an example of a planning instance with the initial trivial path from start to goal configuration. From left to right and top to bottom, the time steps from the start time 0 to the goal time 25 seconds later can be seen. The robot has to move clockwise while the cube moves counter-clockwise.

To get a better impression of the scene, the dimensions of the task are given in Figure 9.2. The rotating cube has walls with outside length 1.5 m and inside length 1.4 m. It rotates with a velocity of 0.1 radians per seconds around the centre of the cube. The robot base and the centre of the first joint respectively is 0.335 m away from the centre, as depicted in the upper right image of Figure 9.2.

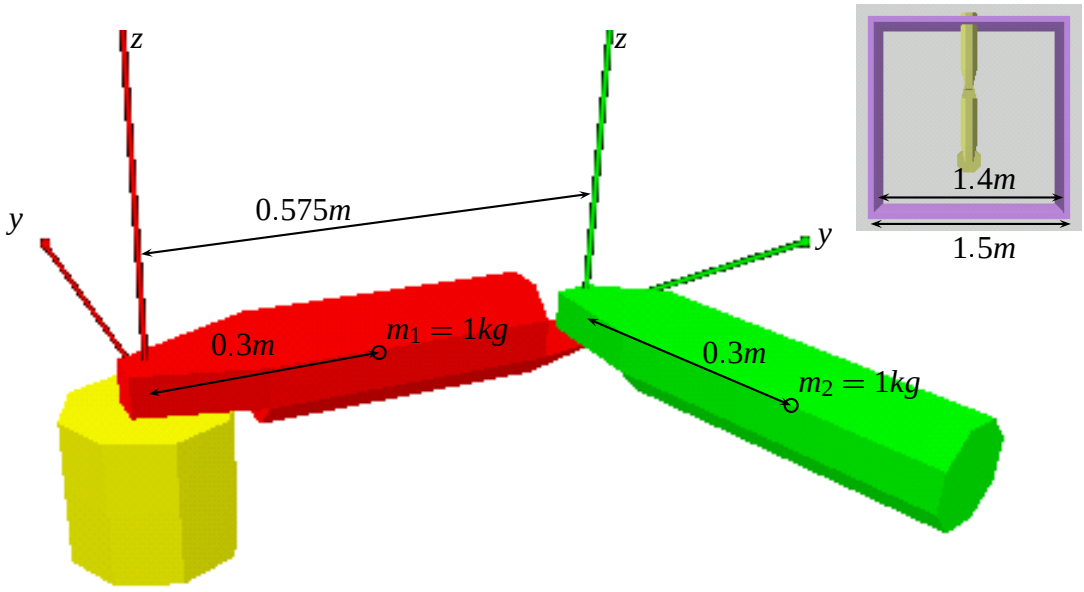

Figure 9.2: *Dimensions of the 2 degrees of freedom example.*

As for the robot dimensions we also refer to Figure 9.2. The length of the first arm is approximately 0.7 m and of the second arm 0.6 m meters. The centre of gravity is located

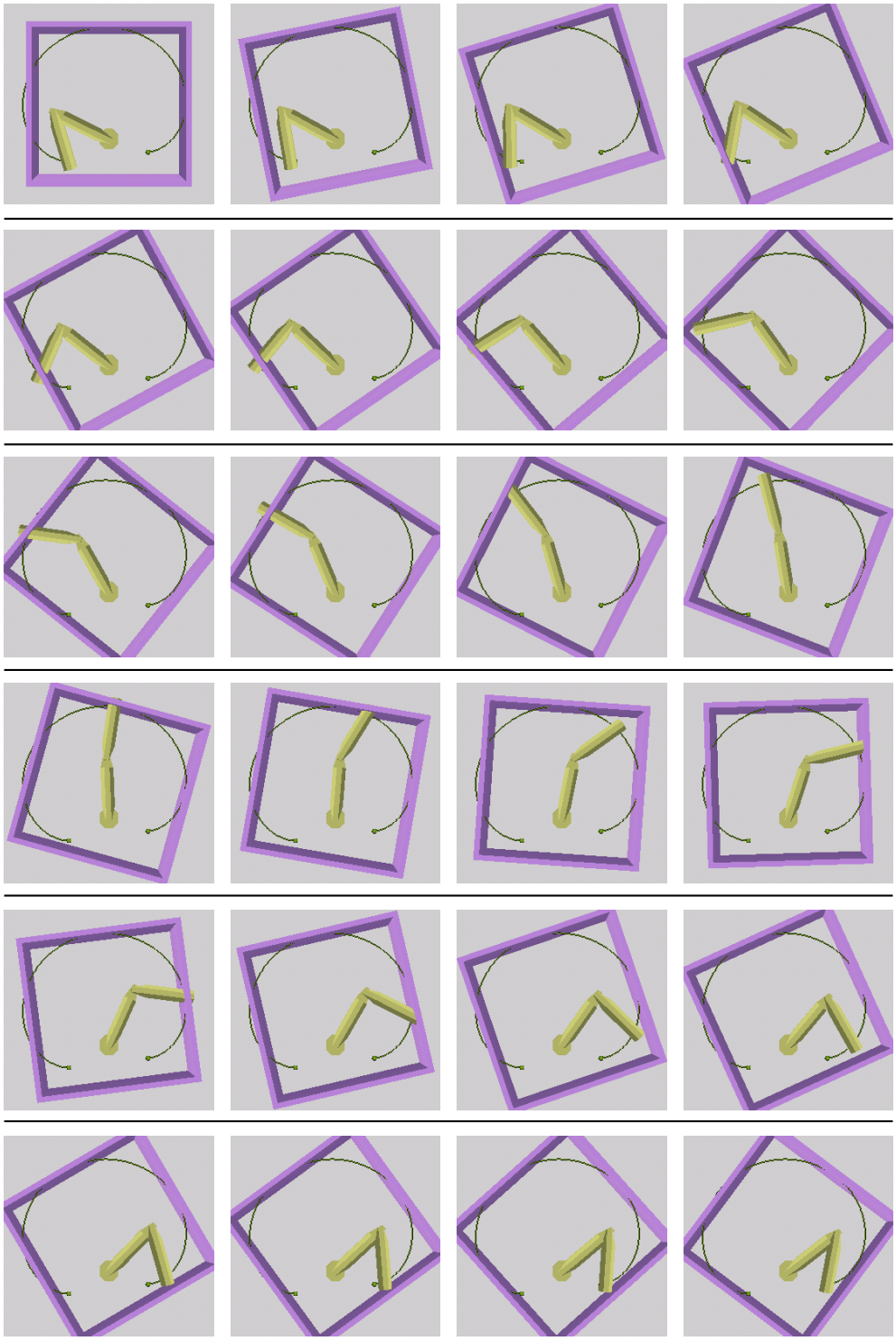

Figure 9.3: *Initial task for the two degrees of freedom robot in a rotating cube. The duration of the movement is 25 seconds.*

at $g_{(1,2)} = (0.3, 0, 0)^T$ m from the joint coordinate system. The weight of each arm is $m_{(1,2)} = 1$ kg and the inertia tensor located at the centre of gravity is $I_{zz} = 30.867\ 10^{-3}$ kg m^2 for each link. The gravity is in the negative z direction of the links. In the upper right corner the scene at time 0 and the robot with joints in zero position can be seen. The values of the joints range from $\upsilon^{\min}_{(1,2)} = -160$ degrees to $\upsilon^{\max}_{(1,2)} = 160$ degrees and the maximal torque is $\theta^{\max}_{(1,2)} = 0.1$ Nm. The start values for the planning task are $\upsilon_s = (63.61, 133.01)$ degrees and the goal values are $\upsilon_g = (-52.05, -138.80)$ degrees.

On pages 131 and 132 two solutions are shown. Figure 9.4 shows a solution with the polynomial point-to-point movement and Figure 9.5 shows a solution with a bang-bang path motion. The solutions show that the main problem for the robot is changing the orientation of its joints. This can only be done if the corner of the cube has the right position. The challenge for the planner is to find the time window and the position of the cube where it can safely perform this operation.

For this problem, we have analysed the running time and whether a solution has been found depending on the chosen precision and step size. In the following tables, the values in boldface implicate a solved solution and the value itself is the running time in seconds. Numbers written in normal font denote the running time for an unsolved solution. All tests have been performed without random movements.

In Table 9.3 results for the modified bang-bang path motion are listed. The search direction is bottom-up and the first joint that improves the rating is taken. From left to right, we have increased the step size for searching a better base point in the configuration space. The precision is increased from top to bottom resulting in less exact ratings for the trajectory.

		step size (mm)				
		1	5	10	20	50
precision	1	**151.05**	**51.52**	**38.24**	**17.42**	**20.64**
	5	**38.84**	**13.38**	**11.03**	**4.25**	**4.55**
	10	**18.07**	**7.03**	**4.46**	**2.24**	**2.50**
	20	**15.00**	**5.81**	**4.80**	**1.67**	**2.05**
	50	**38.22**	**13.51**	**9.74**	**6.02**	**5.09**

Table 9.3: Time used to solve the problem for a modified bang-bang path motion with search direction bottom-up (epsilon = 10^{-8}).

With the modified bang-bang path motion, a solution could be found for all tested values. We note that as the planning time increases, the precision decreases. This is because more ratings have to be calculated. For the step size, the time increases if the steps are too small or too large. This is because if the step is too small, then a lot of moves have to be done before the goal is reached. On the other hand, if steps are too big, then trajectories with good ratings may be missed out. The optimal step size in this example is around 20 mm. The percentage of the running time used for the torque rating is approximately 4 per cent, for the collision rating it is 13 per cent, and for the collision test it is 73 per cent. We would like to note that it is difficult to split the time necessary for

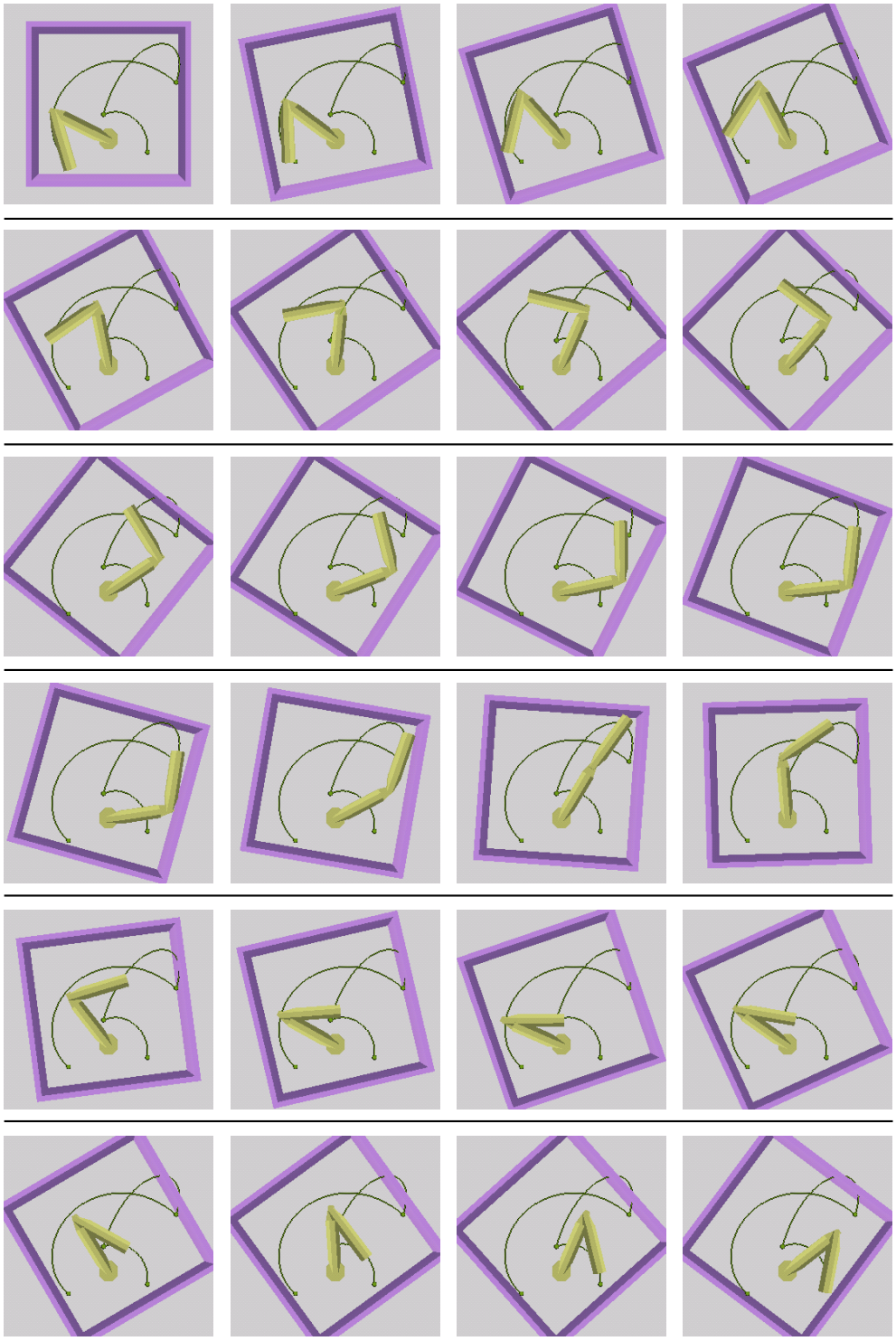

Figure 9.4: *Solved task for the two degrees of freedom robot with a polynomial point-to-point movement.*

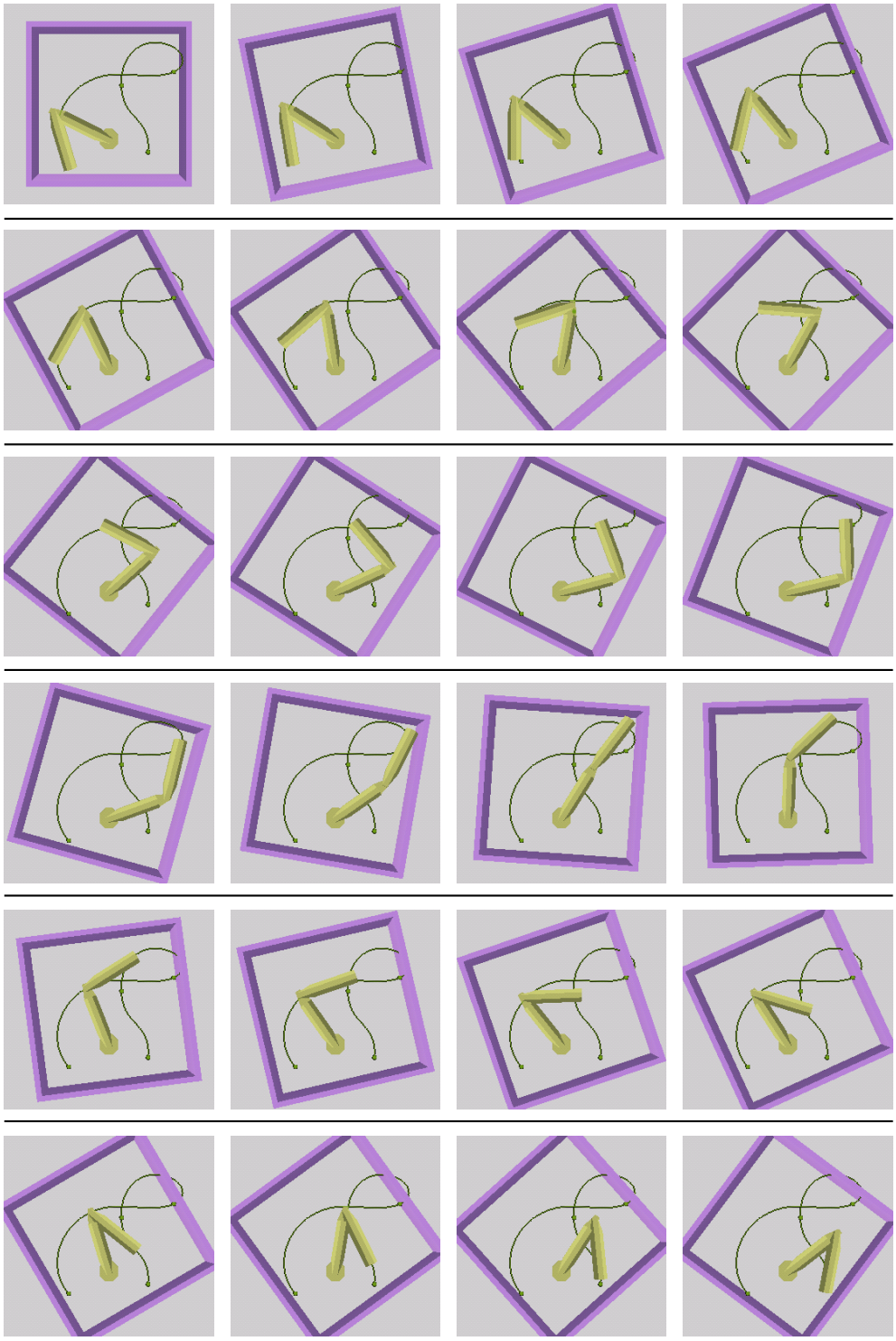

Figure 9.5: *Solved task for the two degrees of freedom robot with a modified bang-bang path motion.*

a collision test and a collision rating since collision rating depends on the collision tests in order to find the points in time where a collision rating should take place.

In another experiment, we increased the maximal torque of the joints to $\theta^{max}_{(1,2)} = 1.0$ Nm. It is not surprising that the task is still solvable for all tested parameter values. The average running time, however, drops from 19.49 seconds to 18.46 seconds. In contrast to this, if the maximal torque is decreased to $\theta^{max}_{(1,2)} = 0.05$ Nm, then the problem is not solvable any more for all tested values of precision and step size. It could not be solved for a precision of 50 and a step size of 50. Here the average computing time is 20.21 seconds.

Let us now turn to the polynomial point-to-point motion. In Table 9.4, the results for the polynomial point-to-point motion are listed. We tested the same parameter values as in the bang-bang path motion. Again, the search direction is bottom-up and the first joint that improves the rating is taken.

		step size (mm)				
		1	5	10	20	50
precision	1	**163.40**	85.62	**35.44**	**25.44**	**17.03**
	5	**37.98**	17.48	**9.71**	**7.17**	**4.34**
	10	**27.11**	**14.4**	**6.84**	**4.54**	**2.69**
	20	**13.51**	10.04	**4.67**	5.48	**2.45**
	50	36.23	14.71	**10.89**	5.34	6.23

Table 9.4: Time used to solve the problem for a polynomial point-to-point motion with search direction bottom-up (epsilon = 10^{-8}).

The results for the polynomial point-to-point movement are not as good as those for the bang-bang motion. In particular, for some combinations of precision and step size, no solution has been found. It turns out that dynamic limits of the robot are more critical in point-to-point movements than in path motions. It is therefore more difficult to find a solution. If we increase the maximal torque to $\theta^{max}_{(1,2)} = 1.0$ Nm and repeat the tests, then with all tested parameter values solutions can be found. The average running time falls from 22.76 seconds to 17.51 seconds. If we decrease the maximal torque to $\theta^{max}_{(1,2)} = 0.05$ Nm then no solutions can be found any more and the average running time rises to 26.52 seconds.

In the next round of experiments, we evaluate how the search strategy (bottom-up, top-down, or best) influences the number of solutions and the running time. We do these tests for both trajectory types and we have generated 100 problem instances with randomly chosen start and goal configurations. The requirements for the randomly chosen examples have been that the start and goal configuration are free and that the start time is in the interval $[0,20]$ and the goal time is in the interval $[40,60]$. The precision value and the step size are 10 and the epsilon value is again 10^{-8}.

In the following tables, the inner tables represent the following values.

The examples which need to be solved are those problems, where the initial start trajectory is not free. We want to distinguish these values, since when testing with randomly

value for all examples	value for all examples which needed to be solved
value for all solved examples	value for all solved examples which needed to be solved
value for all unsolved examples	

generated instances, there is always a percentage of trivial instances where the initial trajectory is already a solution (around 10 per cent).

In Table 9.5, the columns represent: the number of randomly generated problem instances, the percentage of solved instances, the average planning time and the average number of base points in the solution.

Polynomial	Bang-Bang	Bottom-Up	Top-Down	Best	Count		Solved (%)		Time (sec)		Base Points	
x		x			100	68	92	88	3.57	5.23	3.89	4.78
					92	60	100	100	2.54	3.87	3.68	4.58
					8		0		15.42		6.25	
	x	x			100	69	93	90	4.07	5.88	3.86	4.70
					93	62	100	100	2.50	3.72	3.65	4.47
					7		0		24.96		6.71	
x			x		100	68	90	85	4.21	6.17	3.83	4.69
					90	58	100	100	2.94	4.53	3.64	4.55
					10		0		15.71		5.50	
	x		x		100	69	91	87	4.74	6.87	3.97	4.86
					91	60	100	100	2.85	4.33	3.69	4.56
					9		0		23.78		6.78	
x				x	100	68	89	84	6.70	9.86	3.55	4.28
					89	57	100	100	5.00	7.80	3.40	4.19
					11		0		20.53		4.72	
	x			x	100	69	93	90	7.46	10.80	3.60	4.32
					93	62	100	100	4.54	6.81	3.45	4.18
					7		0		46.16		5.57	

Table 9.5: Results for the 2-dof robot in a rotating cube.

We note that planning with the polynomial point-to-point motion is slightly faster than planning with the bang-bang path motion. This is because the calculation of the exact trajectory is more complicated in the path motion and the number of sections that may change if one section is moved is higher. However, with the bang-bang path motion

more problem instances get solved. This is due to the dynamic limitations of the joints of the robot. Since the path motion does not stop at each base point the motion is smoother and hence needs less torque and force.

If the search direction is changed from top-down to bottom-up, we note that the computation time decreases and that more solutions are found. Both effects can be explained by the fact that at the beginning of the planning process, the extent of change is greater if a joint nearer to the base is moved than if one at the top is moved. We also note that taking the best joint in each movement does not increase the number of solved examples. Again, the polynomial point-to-point motion solves fewer instances than the bang-bang path motion. The computation time is higher than in the bottom-up or top-down strategy, since a lot of alternative movements have to be tested before a move can take place.

To summarise, it is apparent that the bottom-up search direction is the best choice here. The modified bang-bang path motion solves more instances, but needs a longer computation time than the polynomial point-to-point motion. Since the difference in computation time is only marginal, we can say that a bottom-up search with the bang-bang path motion is the recommended combination for this scenario.

9.2.2 An Industrial Environment

In this section, we want to take a look at a typical industrial environment. In Figure 9.6 and Figure 9.7 a robot moving between two tables and a machine can be seen. The robot is a six degrees of freedom robot. On the machine, a plate moves on a conveyor belt from the workplace for the machine and the transport place where the robot has to put workpieces. The two figures show the following scenario: the robot picks up a workpiece from table one and moves it to the transport place of the machine (shown in Figure 9.6). Then (after a little while), the robot picks up a workpiece from the transport place of the machine and moves it to table two (shown in Figure 9.7). This scenario is typical for automated manufacturing where machines are loaded and unloaded by robots.

To get a better impression of the dimensions, we give a few sizes for the robot. The x, y, and z extensions of the links given in centimetres relative to their own coordinate system are from base to top: $L_0 : (20,20,30)$, $L_1 : (16,28,16)$, $L_2 : (32.5,15,31)$, $L_3 : (12,34,12)$, $L_4 : (6,6,26)$, $L_5 : (6,3,6)$, $L_6 : (24,6,17)$. For each link the density of the link is $4.5g/cm^2$. The mass, location of gravity, and the inertia tensor were calculated from the model. The maximal torque in the joints are: $J_1 : 0.5$ Nm, $J_2 : 50$ Nm, $J_3 : 50$ Nm, $J_4 : 5$ Nm, $J_5 : 5$ Nm, $J_6 : 0.5$ Nm.

Figure 9.6 shows a solution for the loading part with a bang-bang path trajectory consisting of four base points. The starting point is at time 0 and the goal has to be reached 45 seconds later. For a precision value of 5 and a step size of 10, the trajectory is found after 1.75 seconds using the bottom-up search direction. In Figure 9.7, a solution for the transport from the machine to the second table can be seen. Here the computation time was 1.86 seconds.

For the rest of this experiment, we focus on the robot's motion for the loading part of this scenario, that is, where the robot moves a workpiece from table one to the transport place of the machine (Figure 9.6). In this example it is not possible to take a large precision value because start and goal configurations are quite near the obstacles. We have

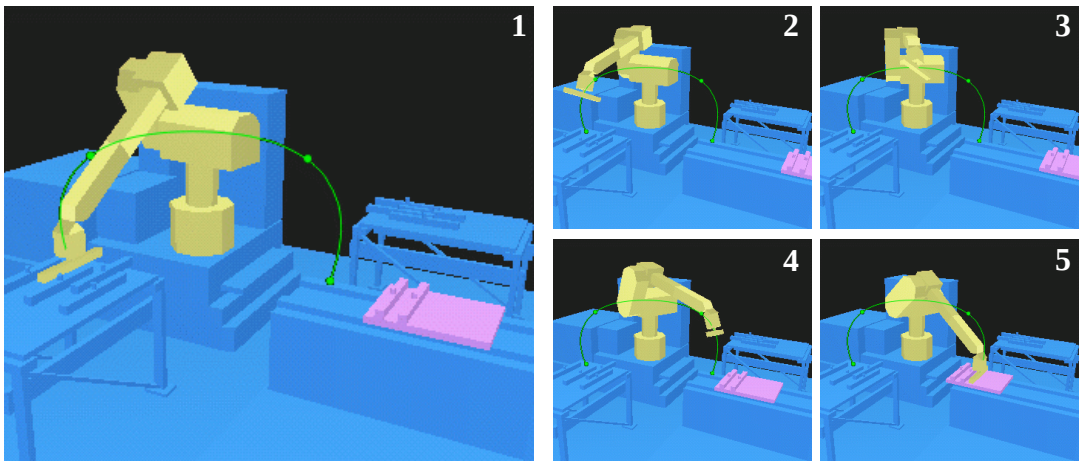

Figure 9.6: *Loading the machine: a workpiece is moved from table one to the conveyor belt*

run the experiment with a precision of 5 at different step sizes, different search directions, with and without random joint selection, and two different trajectory types (polynomial point-to-point motion and bang-bang path motion). Table 9.6 shows the computation times in each case.

Polynomial	Bang-Bang	Bottom-Up	Top-Down	Random	step size (mm)				
					1	5	10	20	50
x		x			1.98	2.12	1.56	1.41	1.71
x		x		x	13.10	8.72	18.79	7.80	4.08
x			x		58.29	27.90	23.42	29.54	25.86
x			x	x	24.37	10.97	12.41	14.02	14.68
	x	x			4.32	20.97	1.75	1.33	1.59
	x	x		x	23.73	17.14	12.52	10.73	5.16
	x		x		56.83	50.26	46.05	21.11	8.18
	x		x	x	63.48	24.22	15.78	15.20	5.97

Table 9.6: Time used to solve the machine loading task for different parameter choices.

The examples with the random joint strategy have been done 50 times and the average values are taken. Again we note that the best joint search direction, as far as computation time is concerned, is bottom-up in the hierarchy of the joints (there is only one exception for the bang-bang motion with step size 5). Moreover, we can observe that the bottom-up search direction is the best choice for both the polynomial point-to-point and the modified bang-bang path trajectory for this task. This is because there is a lot of free configuration space and the robot can easily find a free trajectory by moving the joints nearer towards

the base. On average, it does not decrease computing time if we chose the starting joint randomly, but we would like to mention that for the polynomial, bottom-up, random case the best time was 0.51 and the worst time was 51.12 seconds. If the search direction is changed to top-down, then the lower bound stays the same, but the upper bound rises to 81.6 seconds. For the bang-bang trajectory we get a lower bound of 0.53 seconds, and an upper bound of 151.15 seconds for the bottom-up strategy and 152.19 seconds for the top-down strategy respectively.

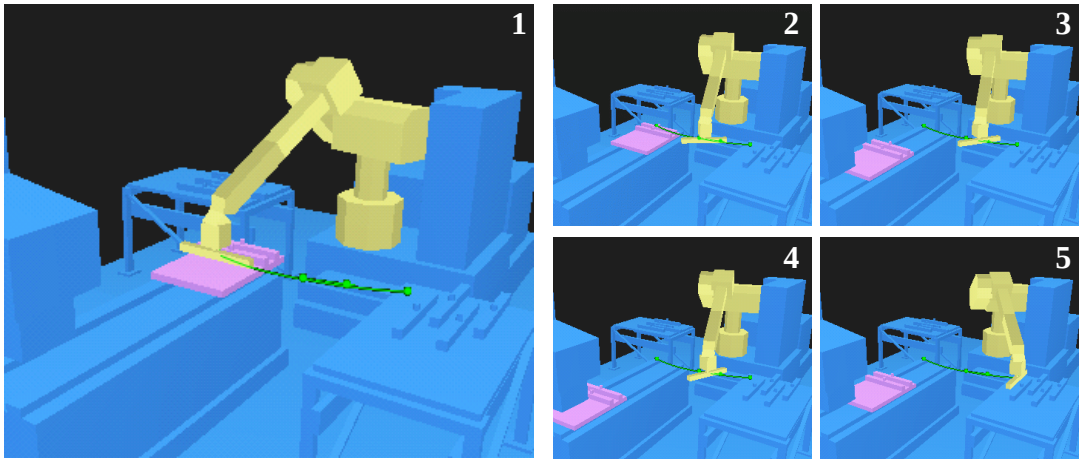

Figure 9.7: *Unloading the machine: a workpiece is moved from the conveyor belt to the second table*

Another parameter that we want to analyse is the epsilon value. For the bottom-up search direction and the polynomial and bang-bang motion we changed the epsilon values and the step sizes, leaving the precision unchanged at 5. In Table 9.7 the values for the polynomial point-to-point motion and in Table 9.8 the results for the modified bang-bang trajectory are listed.

		step size				
		1	5	10	20	50
epsilon	10^{-8}	**1.97**	**2.11**	**1.56**	**1.41**	**1.70**
	10^{-3}	**1.98**	**2.11**	**1.56**	**1.41**	**1.69**
	1	8.27	**2.01**	**1.56**	**1.41**	**1.69**
	5	8.29	**2.00**	**1.55**	**1.41**	**1.70**
	10	8.29	8.26	**1.55**	**1.41**	**1.81**

Table 9.7: Time used to solve the machine loading example for a polynomial point-to-point motion with search direction bottom-up.

Apparently, larger epsilon values should only be used with larger step sizes. This is because a smaller step size normally results in small changes in the rating. Larger epsilon values tend to result in shorter computation times.

		step size				
		1	5	10	20	50
epsilon	10^{-8}	**4.31**	**20.82**	**1.74**	**1.32**	**1.60**
	10^{-3}	**4.33**	**14.95**	**1.72**	**1.31**	**1.61**
	1	13.65	**1.94**	**1.64**	**1.32**	**1.62**
	5	8.05	**1.93**	**1.65**	**1.32**	**1.60**
	10	8.05	7.66	**1.64**	**1.31**	**1.95**

Table 9.8: Time used to solve the machine loading example for a modified bang-bang path motion with search direction bottom-up.

9.2.3 Benchmarks for Static Environments

In this section, we compare our algorithm with existing planning algorithms. Since no other implementations for planners exist that are able to plan in time-varying environments we restrict our experiments to the static case. It is difficult to compare algorithms since there exist no commonly accepted benchmarks for robot motion planning, especially not for motion planning in time-varying environments or planning with constraints on robot dynamics. We picked a set of static benchmarks and two existing planning algorithms for which planning times for the benchmarks have been published.

At the Institute for Process Control and Robotics (IPR) of the University of Karlsruhe, a parallel motion planning algorithm for industrial robot arms with six degrees of freedom in an on-line 3D environment has been developed [Wörn et al., 1998]. The method is based on the A*-search algorithm and works in an implicitly discrete configuration space. The idea is to map each considered configuration during the A*-search expansion process to one processor. The collisions test itself is done by fast, hierarchical distance computations [Henrich and Cheng, 1992]. In principle, this approach should also be useful in time-varying environments. However, the results published in [Wörn et al., 1998] only cover the static case.

For the experimental results, the benchmarks found in [Wörn et al., 1998] are taken. These benchmarks are based on benchmarks for a two degrees of freedom planning problem originally presented in [Hwang, 1996]. The benchmarks consist of five problems to solve with a Puma260 robot which is fixed to the ceiling. The Puma260 robot itself has six joints. There are no moving obstacles since the benchmarks have been developed for static environments.

The other planning algorithm that we would like to include in our comparison is the BB-method [Baginski, 1999], which was developed at the Institute for Real-time Systems and Robotics at the University of Munich. In [Baginski, 1999], results for the above mentioned benchmarks can be found as well. Again, the BB-method neither considers dynamic limits of the robot nor time-varying obstacles.

In Figure 9.8 four benchmarks, one in each row, are depicted. Each row shows four positions of the robot on the initial trajectory from start to goal configuration.

The first example is a simple problem where the robot has to pass one small obstacle (simple-task). In the second example, the robot moves from one box to another (star-

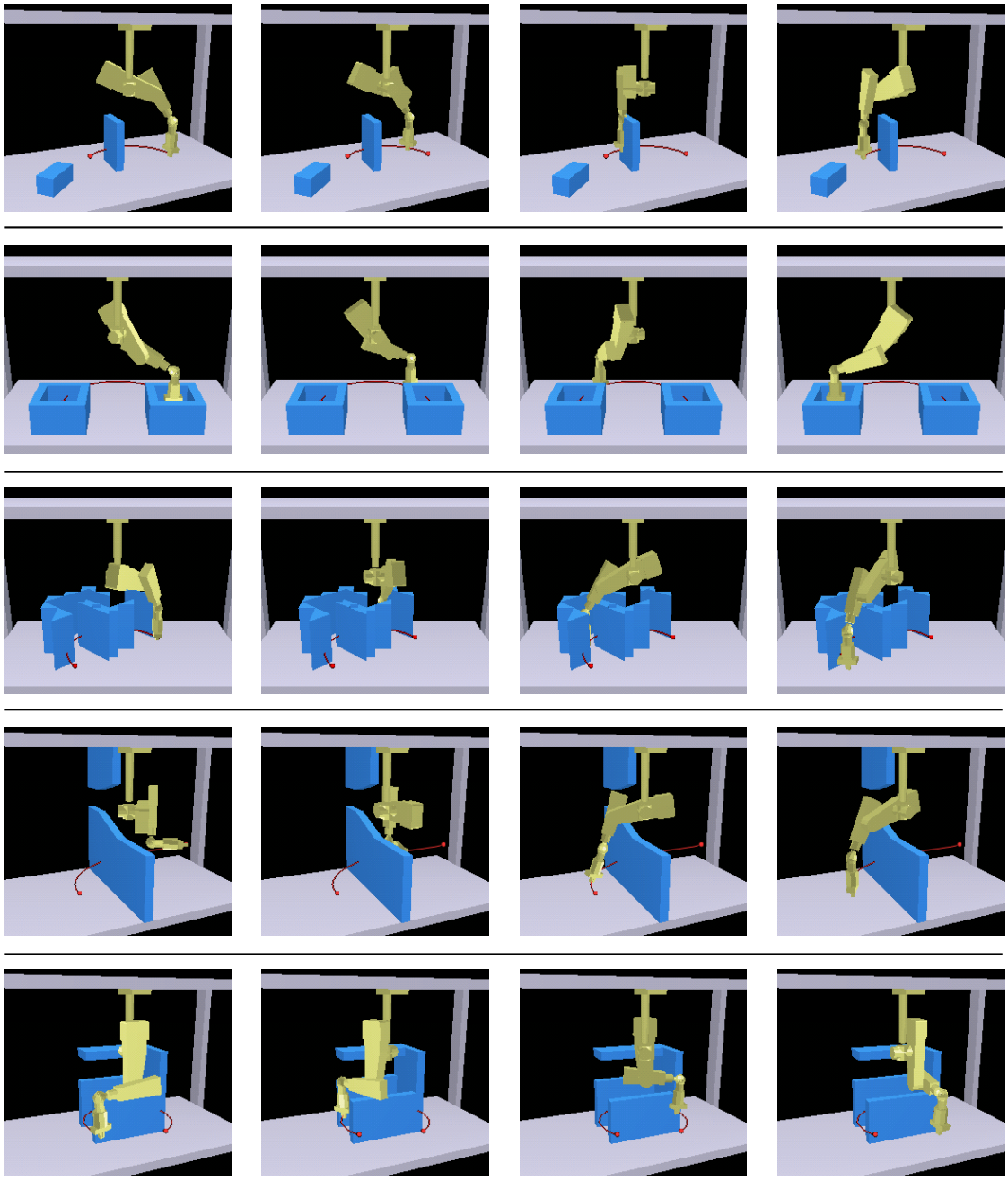

Figure 9.8: *Benchmark tasks with initial trajectories. From top to bottom: simple-task, star-task, detour-task, bottleneck-task, and trap-task.*

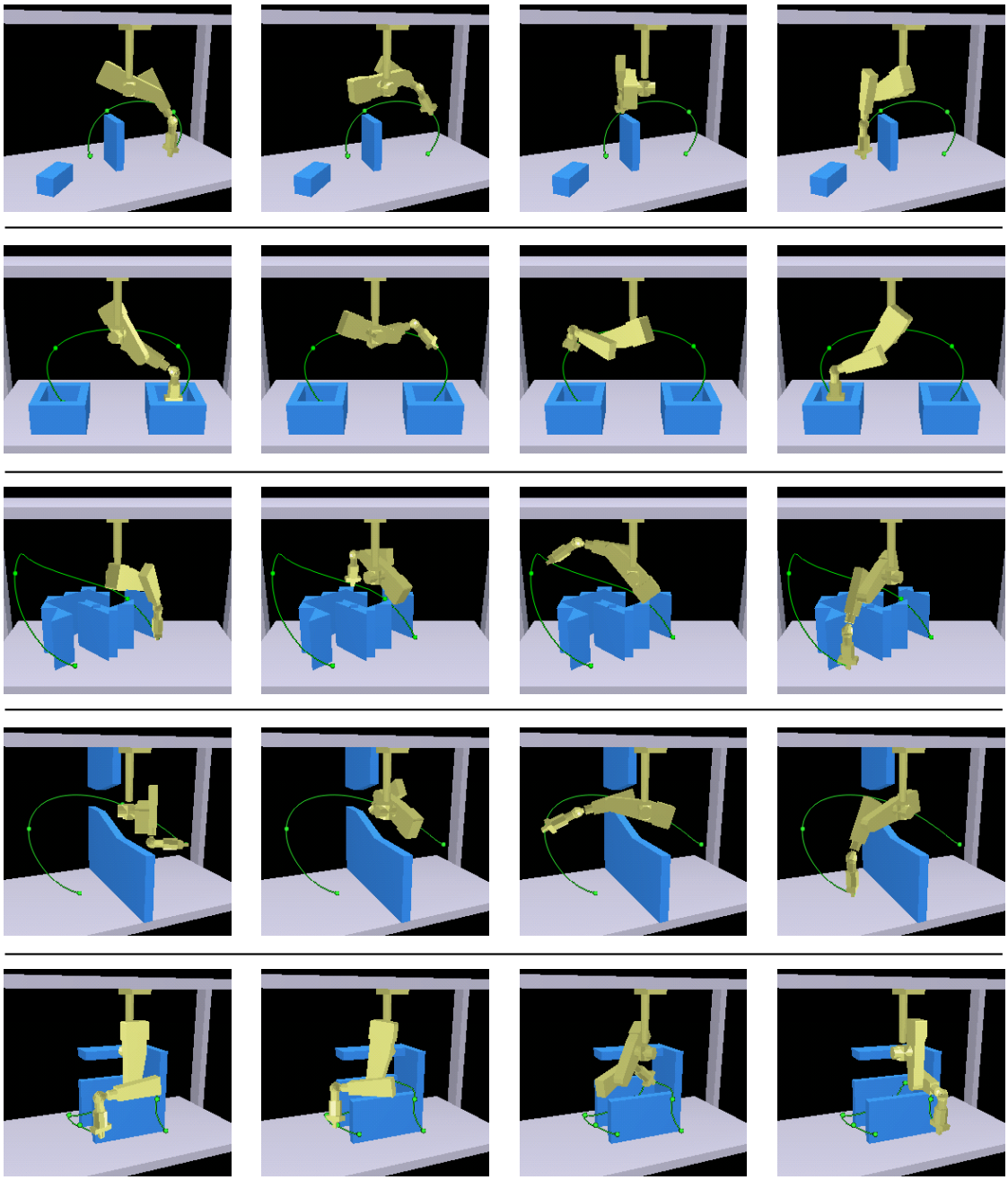

Figure 9.9: *Benchmark tasks with solutions. From top to bottom: simple-task, star-task, detour-task, bottleneck-task, and trap-task.*

task). In the third row, the robot has to find its way over the obstacles and not through the obstacles (detour-task). In the fourth example, the robot has to pass a small gap in the wall (bottleneck-task). The final example is quite difficult as the robot nearly gets trapped in the horseshoe shaped obstacle (trap-task). The travel time for each task is 60 seconds.

Figure 9.9 shows possible solutions to the problems. Again each row shows four positions of the robot for one problem.

For the two types of trajectories, the polynomial point-to-point motion and the modified bang-bang path motion, we are interested in the computing time. All of the following tests have been done for bottom up and best search (without random start joint).

Let us now turn to results for the simple-task. In Table 9.9, the left column lists the results for the polynomial trajectory type and the right shows the results for the bang-bang motion.

Polynomial

Bottom-Up

precision \ step size	5	10	20	50
5	3.82	2.93	2.23	1.38
10	1.74	2.14	1.63	1.02
20	1.35	1.65	1.28	0.78
50	1.22	1.00	1.37	0.79

Best

precision \ step size	5	10	20	50
5	9.42	9.34	7.25	6.12
10	7.43	6.53	5.11	4.68
20	6.91	5.96	4.56	3.98
50	6.96	6.06	5.12	2.13

Bang-Bang

precision \ step size	5	10	20	50
5	4.42	3.32	2.08	1.29
10	3.11	2.38	1.55	0.88
20	1.76	1.96	1.31	0.77
50	1.30	1.68	1.46	0.78

precision \ step size	5	10	20	50
5	12.10	10.75	8.23	6.43
10	7.76	7.23	5.34	4.06
20	6.52	6.25	4.55	3.42
50	8.78	4.76	3.68	2.74

Table 9.9: Time used to solve the **simple**-task benchmark for a polynomial point-to-point motion (left) and the modified bang-bang path motion (right) with search direction bottom-up (top) and best (bottom).

The planning algorithm solved the problem for all parameters with an average time of **1**.**80** for the bottom up search and **6**.**26** for taking the best joint. In comparison to our results, the BB-method needed **0**.**06** seconds (Intel Pentium II, 266MHz) and the parallel A*-search required approximately **2** seconds on one processor (Intel Pentium, 133MHz, 64 MByte Main Memory).

The next example is the star-task where the robot has to grasp from one box into another. In Table 9.10 the left tables shows the results for the polynomial motion, whereas the right tables present the bang-bang trajectory results.

The star-task has been solved by our planner for all tested parameter values. The average planning time for the bottom-up search was **5**.**70** seconds and for taking the best joint **11**.**98**. The computation time for the BB-method was **0**.**59** seconds, and the parallel A*-search algorithm took approximately **25** seconds on one processor and less than **5** seconds on eight PC's.

Polynomial

Bottom-Up

precision \ step size	5	10	20	50
5	2.35	2.40	3.52	2.11
10	2.63	2.26	1.98	1.48
20	5.23	4.21	4.26	2.36

Best

precision \ step size	5	10	20	50
5	19.44	11.62	11.16	8.70
10	13.18	9.16	7.82	7.06
20	12.55	8.55	6.53	7.69

Bang-Bang

Bottom-Up

precision \ step size	5	10	20	50
5	2.82	2.78	1.80	1.84
10	1.36	3.47	2.11	2.62
20	4.49	9.33	53.80	15.23

Best

precision \ step size	5	10	20	50
5	16.34	8.18	8.20	9.56
10	15.41	12.95	10.13	7.81
20	31.16	13.10	24.24	6.87

Table 9.10: Time used to solve the **star**-task benchmark for a polynomial point-to-point motion (left) and the modified bang-bang path motion (right) with search direction bottom-up (top) and best (bottom).

The third example is the detour-task. Here the problem for the planner is to find the path over the obstacles, instead of moving right through the obstacles. Table 9.11 gives the computing times for the tested parameters.

Polynomial

Bottom-Up

precision \ step size	5	10	20	50
5	**33.34**	**11.44**	**29.68**	**12.79**
10	**23.64**	**23.91**	**20.65**	**12.84**
20	**18.44**	**33.00**	**68.10**	**9.84**
50	18.83	36.68	15.23	**2.16**

Best

precision \ step size	5	10	20	50
5	**46.24**	**40.56**	**35.87**	**30.78**
10	**34.61**	**28.67**	**33.74**	**19.77**
20	**26.30**	**37.27**	**39.73**	**17.16**
50	66.93	**102.68**	39.85	**10.18**

Bang-Bang

Bottom-Up

precision \ step size	5	10	20	50
5	**5.87**	**5.54**	**44.92**	**20.97**
10	**7.85**	**3.92**	**23.27**	**4.56**
20	56.91	**64.25**	**35.12**	**14.58**
50	**3.78**	**3.38**	**2.84**	**4.49**

Best

precision \ step size	5	10	20	50
5	**115.13**	**34.61**	**29.93**	**29.75**
10	**60.31**	**24.88**	**21.55**	**19.84**
20	**52.25**	**20.97**	**18.01**	**14.38**
50	98.61	**14.23**	**11.62**	**11.12**

Table 9.11: Time used to solve the **detour**-task benchmark for a polynomial point-to-point motion (left) and the modified bang-bang path motion (right) with search direction bottom-up (top) and best (bottom).

Unfortunately not all tested parameter values resulted in a solution. The robot may get stuck in the labyrinth if the step size is too small or the precision value is too large. (Please note that if random moves are allowed then the situation improves and more solutions are found.) The BB-method requires **0.70** seconds and the parallel A*-search algorithm takes about **110** seconds on one processor and still **20** seconds on eight processors.

Table 9.12 shows the results for the bottleneck-task using the polynomial point-to-point trajectory on the left side. On the right side, the times for the modified bang-bang path motion can be seen.

Polynomial

Bottom-Up

precision \ step size	5	10	20	50
5	**148.84**	**147.04**	479.04	140.72
10	**427.77**	**82.84**	219.42	**220.74**
20	82.59	75.64	97.29	84.57

Bang-Bang

precision \ step size	5	10	20	50
5	**59.76**	**109.88**	**35.96**	**62.69**
10	**82.04**	**52.16**	**36.37**	102.29
20	**77.65**	**388.94**	482.07	83.36

Best

precision \ step size	5	10	20	50
5	**78.20**	**48.45**	**40.67**	**57.55**
10	**51.78**	**32.99**	**36.98**	**46.07**
20	**96.33**	105.34	48.29	**29.16**

precision \ step size	5	10	20	50
5	**127.47**	**62.23**	**48.13**	**36.25**
10	**84.76**	**46.10**	**36.59**	**25.67**
20	174.81	84.69	60.45	**34.68**

Table 9.12: Time used to solve the **bottleneck**-task benchmark for a polynomial point-to-point motion (left) and the modified bang-bang path motion (right) with search direction bottom-up (top) and best (bottom).

We observe that the problem does not get solved if the precision value is too large. This is because the robot has to find its way through the small gap in the wall. The same observation can be made for the step size. If the step size is large, the robot will miss the gap in the wall.

The calculation time is relatively high compared with the result of the BB-method. In [Baginski, 1999], the computation time for this problem is given as **0.65** seconds. To explain this discrepancy, we would like to mention that the BB-method is, at the time being, presumably the best planner for static environments. However, it does not take into account dynamic limitations of the robot or moving obstacles. For the same problem, the parallel A*-search in [Wörn et al., 1998] needs about **23** seconds on one processor (Intel Pentium, 133MHz, 64 MByte Main Memory) and about **5** seconds on eight PC's.

To demonstrate the advantages of our method, we try to compare the dynamic behaviour of the solutions found with the BB-method and with our method. To this end, we distribute base points along the path found with the BB-method. The overall travel time is the same as in our solutions. In Figure 9.10 on the left side the solution with the BB-method and on right side one of our solutions can be seen (precision is 10, step size is 20, search direction is bottom up).

The green line represents our solution and the red line the optimised path found with the BB-method. The path length is optimised in the configuration space. It shows that the green lines are closer to the robot's base. Therefore the robot needs less torque to hold its upper and lower arm. In Figure 9.11, the torque needed over the time is depicted. We only show the values for the second and the third joint counting from the base. These are the joints that need the most force in order to hold the rest of the robot.

On the left side, the values for the detour-task can be seen. The black curves represent

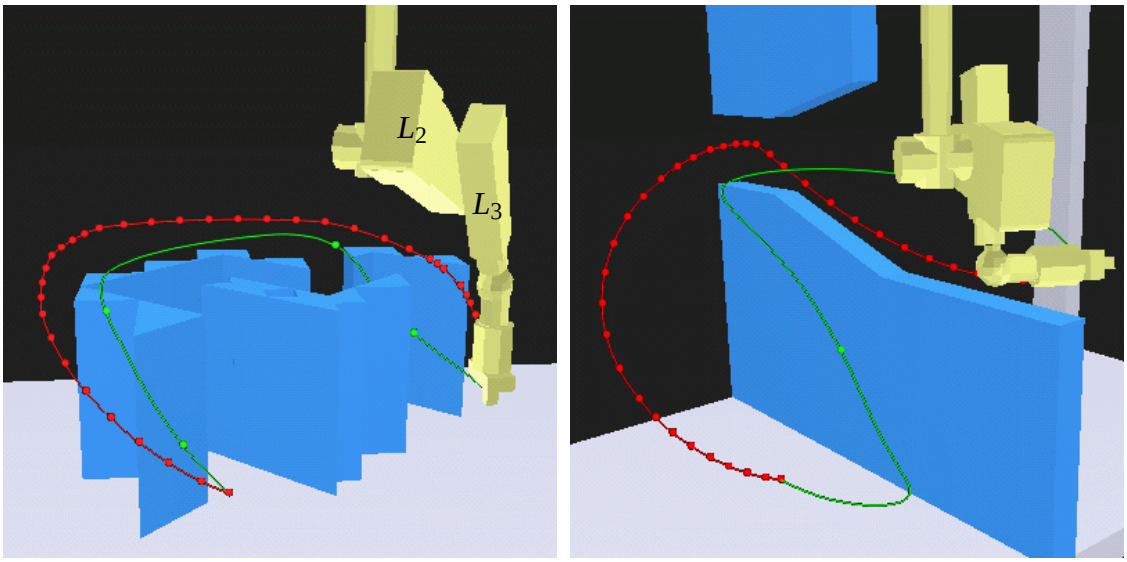

Figure 9.10: *Comparison of trajectories found with the BB-method and our method. The red outer path represents the path computed with the BB-method, the green inner trajectory is generated using our algorithm.*

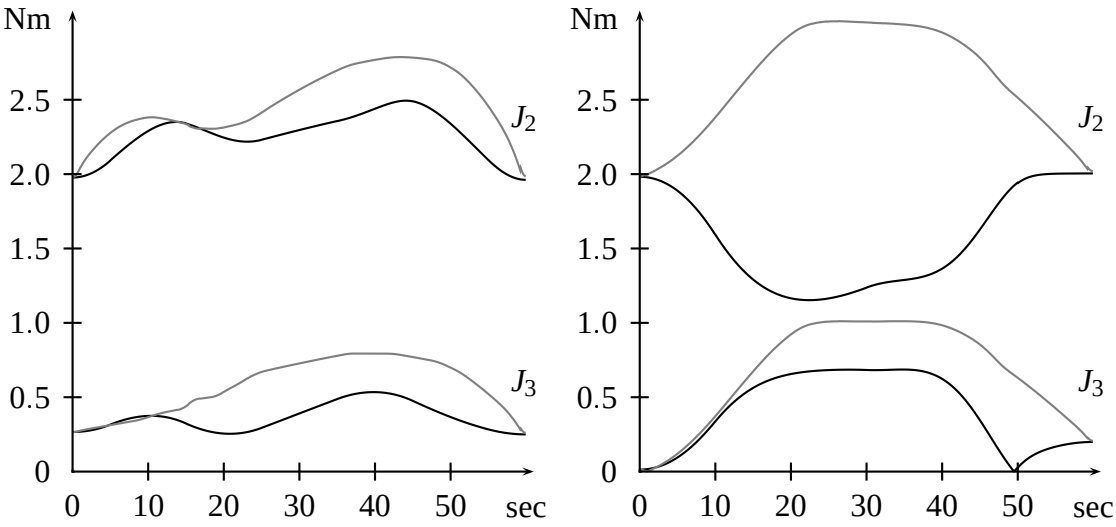

Figure 9.11: *Torque of the second and third joint. The left side shows the torques for the detour-task and the right side the torques of the bottleneck task. The black curves represent the values for our solution and the grey curves the solution of the BB-method.*

the values for our solution and the grey curves the solutions of the BB-method. The upper curves give the values of joint two. This is the joint which has to carry the most weight. On the right side, the curves are given for the bottleneck-task. It is obvious that our solution needs less torque. This is because our trajectories bring the centre of gravity of the links nearer to the robot's base.

For the last benchmark example, computing times are not given in either [Baginski, 1999] or in [Wörn et al., 1998]. We include our results for this benchmark here for the

sake of completeness. If we allow no random movements, this problem is hard to solve for our planner (its only solvable if the step size is large enough such that the robot is not caught in the trap). For this last example we allow 10 random movements during the planning. The times for the last example are presented in Table 9.13. We have only tested the bottom-up search direction.

Polynomial

Bottom-Up precision	step size 5	10	20	50
5	**1152.31**	**2078.64**	**479.04**	**1054.95**
20	**229.69**	**2145.14**	**172.47**	**29.60**
50	**399.67**	5150.69	**181.46**	**114.12**

Bang-Bang

Bottom-Up precision	step size 5	10	20	50
5	**739.02**	**1254.81**	**72.05**	**30.99**
10	**104.48**	**640.39**	**38.57**	**8.58**
20	**535.40**	**251.43**	**104.02**	1495.20

Table 9.13: Time used to solve the **trap**-task benchmark for a polynomial point-to-point motion (left) and the modified bang-bang path motion (right) with search direction bottom-up.

These benchmark results reveal that our planner has difficulties to compete with the fast BB-method as far as running time is concerned. But as soon as aspects of real world robots get important, namely torque and force, the solutions found by our planner are better. And in situations where force and torque are critical, the solutions found by a planner that only considers collisions may not even be traceable.

Compared to the parallel A*-search method, our planner is quite good. The computation times of our planner and the times given for the A*-search method are similar, taking into account the differences in clock rate and number of concurrent processors. Moreover, the A*-search planning approach is not applicable to higher dimensional problems, as it needs to distribute possible neighbours in the configuration space to the available processors. In the experiments described in [Wörn et al., 1998], the configuration space has been normalised such that the actual number of dimensions is in fact reduced by one. This is possible because in the above benchmarks the impact of the outermost joint is negligible.

9.2.4 RX90 with a Heavy Hand

In this section, we want to show an example where it is not the collision but the dynamic limitation of the robot that plays the important role. To this end, we took a geometry model of the RX90 robot of Stäubli (see Figure 9.12) and we set the masses of the robot's links in a very special way. We made the hand so heavy that there exist unstable robot configurations, that is, configurations that the robot cannot maintain because it is not able

to compensate for gravity. In other words, in some configurations, the maximal possible torques in the second and the third joint are too small to carry the rest of the robot.

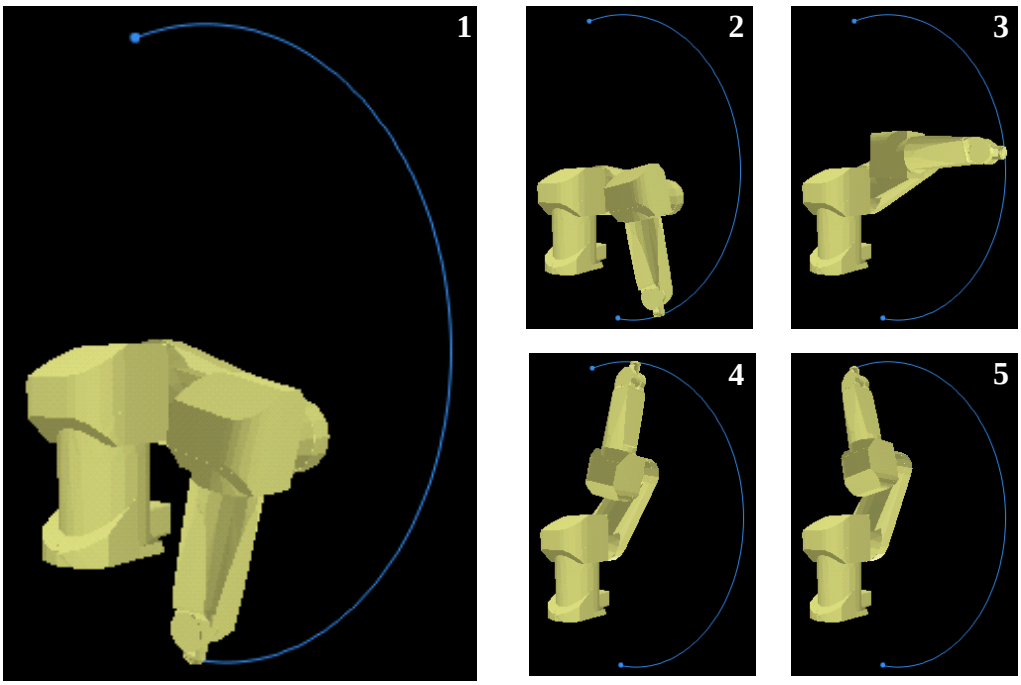

Figure 9.12: *Initial task for the RX90 robot with a heavy top.*

Let us take a closer look at Figure 9.12. The robot has to move its hand from the bottom to the top. The initial trivial trajectory is collision free and would be valid if we would only consider collisions. However, when tracing this trivial trajectory, a position is reached where the robot holds its arm stretched out horizontally. Due to the given dynamic joint limits and the heavy hand, this position cannot be hold by the robot's joints. As a consequence, the initial trajectory is unfree.

Let us take a look at the next two figures. In Figure 9.13, a solution computed by our planner is shown that uses the modified bang-bang path motion. Now the robot uses its own mass to accelerate its hand and it keeps the tool close to its centre of mass in order to reduce the torques that it needs to hold its own links. We would expect a similar movement from an athlete lifting a heavy dumb-bell.

The same problem solved with the polynomial point-to-point motion is depicted in Figure 9.14. Again the upper arm is kept near to the centre of mass of the robot.

We have run this task for different step sizes (1, 5, 10, 20, 50) and precision values (1, 5, 10, 20, 50). With the bang-bang path motion the average computation time is **3** seconds and with the polynomial point-to-point motion this time is **2**.**16** seconds. The task was solved for all tested parameter values.

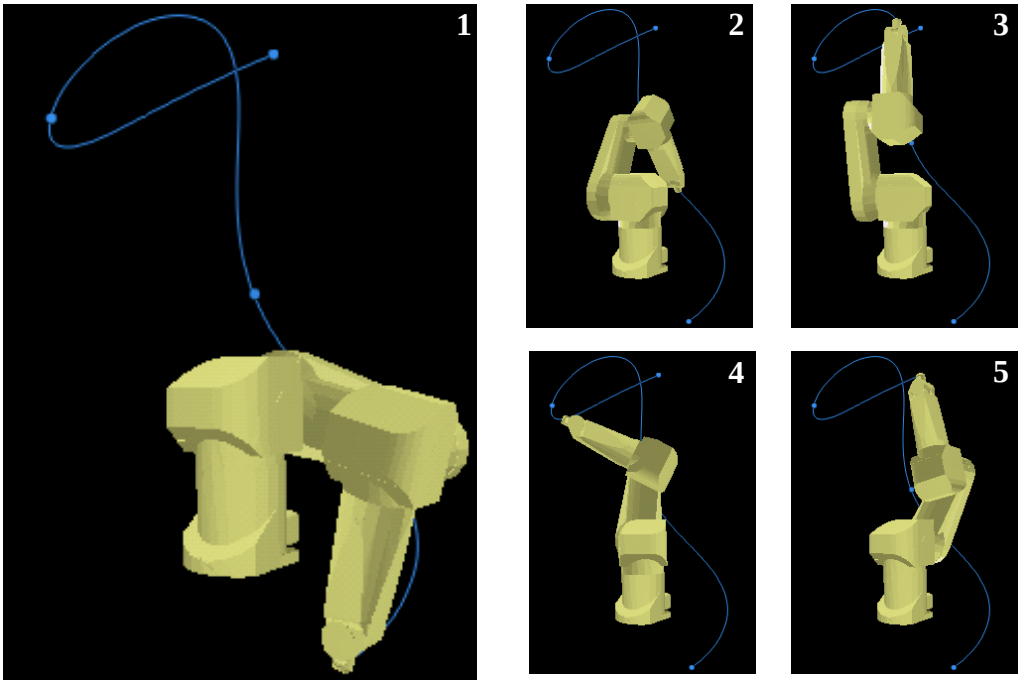

Figure 9.13: *RX90 task solved with a bang-bang path motion.*

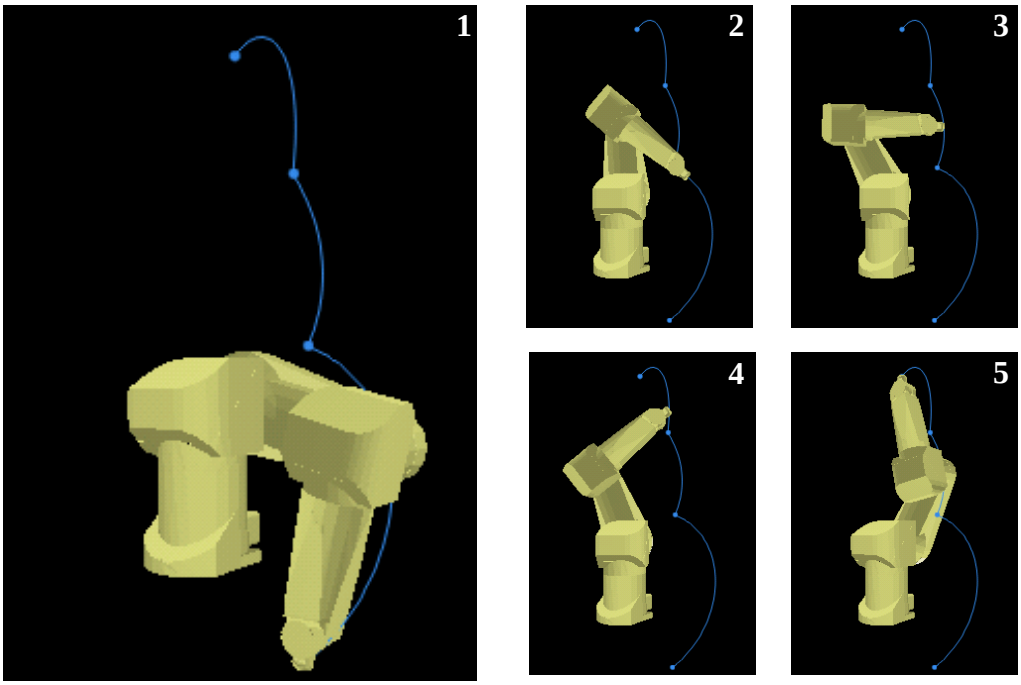

Figure 9.14: *RX90 task solved with a polynomial point-to-point motion.*

9.2.5 Sliding Door

The following example was introduced in [Baginski, 1999] as a static example with narrow passages and it was used for comparisons with the randomised path planner (RPP) of Challou and Gini [Challou et al., 1998]. We have extended this example to a problem with a time-varying environment (see Figure 9.15)

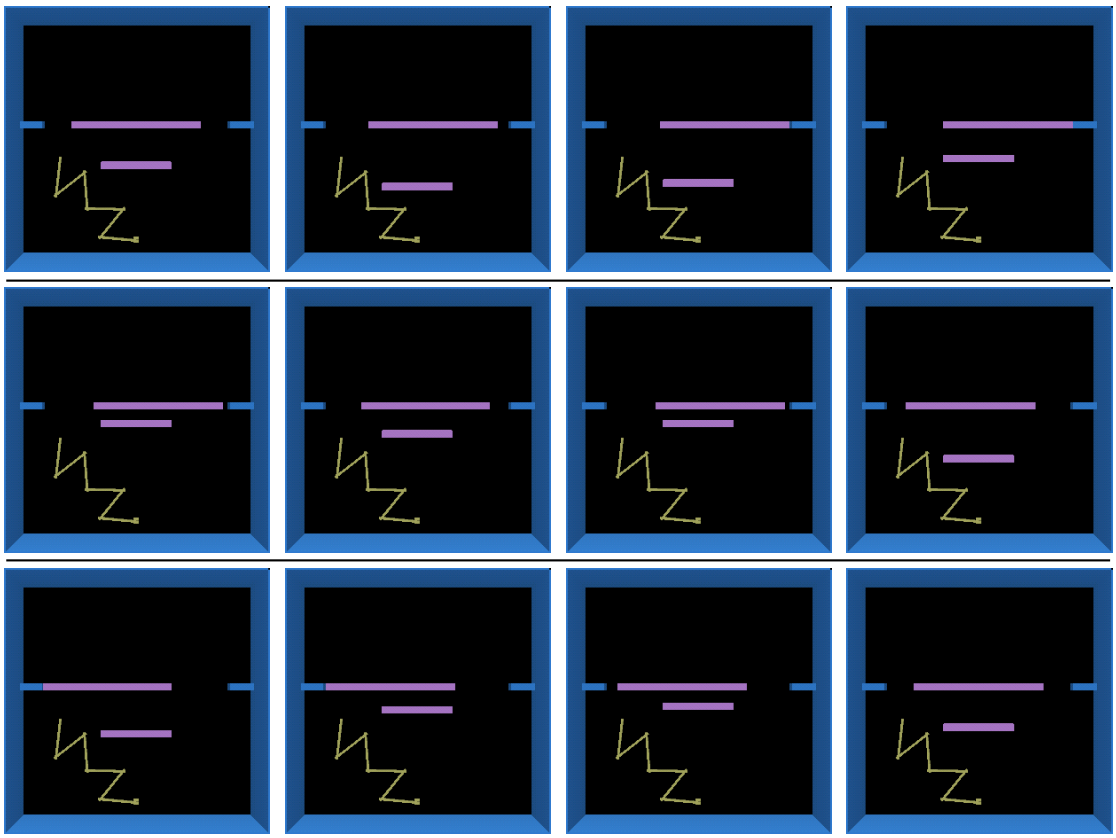

Figure 9.15: *Movement of the time-varying obstacles in the sliding door environment.*

In the original example, the upper bar is fixed in the middle, as in the upper left picture. The lower bar is fixed quite near to the robot's base, as in the second picture from the left in the top row. The robot itself is a six degrees of freedom robot. In the figure, the image sequence from left to right and from top to bottom shows one cycle of movement of the time-varying obstacles. The cycle repeats every 80 seconds. We let the upper bar become a sliding door creating a gap to its left and right in an alternate fashion, where the robot may reach through. The cycle of the second bar is twice as fast as the cycle of the upper bar and it moves vertically away from the robot's base towards the sliding door and back again.

Before we present our test results, we would like to give the exact dimensions of this example. We believe that this is a good benchmark for other planners because the underlying problem is not trivial and it is simple to rebuild. We will not give the height of the obstacles since their height is not important as long they are high enough in order to

be an obstacle for the robot. In Figure 9.16 the scene can be seen from above.

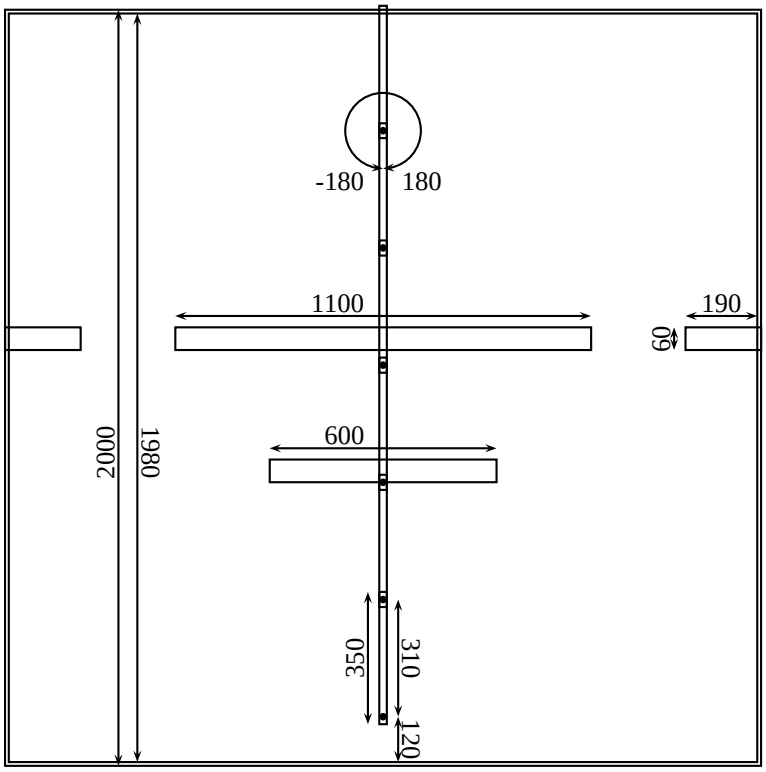

Figure 9.16: *Dimensions of the sliding door environment. All values are given in millimetres or degrees.*

The scene is mirrored vertically. The inner bounds of the box are 1980 mm x 1980 mm. All robots links and joints are identical. The overall dimension of one link is 20 mm x 350 mm x 20 mm. The density is given by 0.001 g/mm^2. This results in a mass of 140 g if we assume an equal distribution of the mass. The centre of gravity is in the middle of the link, and the inertia tensor is $I_{zz} = 140/12(20^2 + 350^2)$ gmm$^2 \approx 1433833$ gmm^2. The robot in the figure is in zero position. The minimal and maximal values the joints may have are -180 and 180 degrees. (As a consequence, it is essential to test for self-collisions of the robot.) The maximal torque in each joint is 0.07 Nm.

Finally, we have to specify the movements of the two time-varying obstacles. The larger bar changes its position from left to right with the following function defined over time $250.0\sin(\pi/40t)$ mm and the smaller bar changes its position up and down with $-200.0\sin(\pi/20t)$ mm.

In Figure 9.17, the solution for a task can be seen, where the robot has to move out of the left gap into the right gap. The start configuration for the robot's joints is $\vec{\upsilon}_s = (13, 73, -65, 11, -30, 30)$ (bottom-top) and the goal configuration is mirrored $\vec{\upsilon}_g = (-13, -73, 65, -11, 30, -30)$. The duration time is 40 seconds. For this environment, we only tested the modified bang-bang trajectory and the search direction bottom-up in the robot's joints, because this combination turned out to be the most successful one. To solve this task, our planner needed approximately 20 seconds with a precision value of

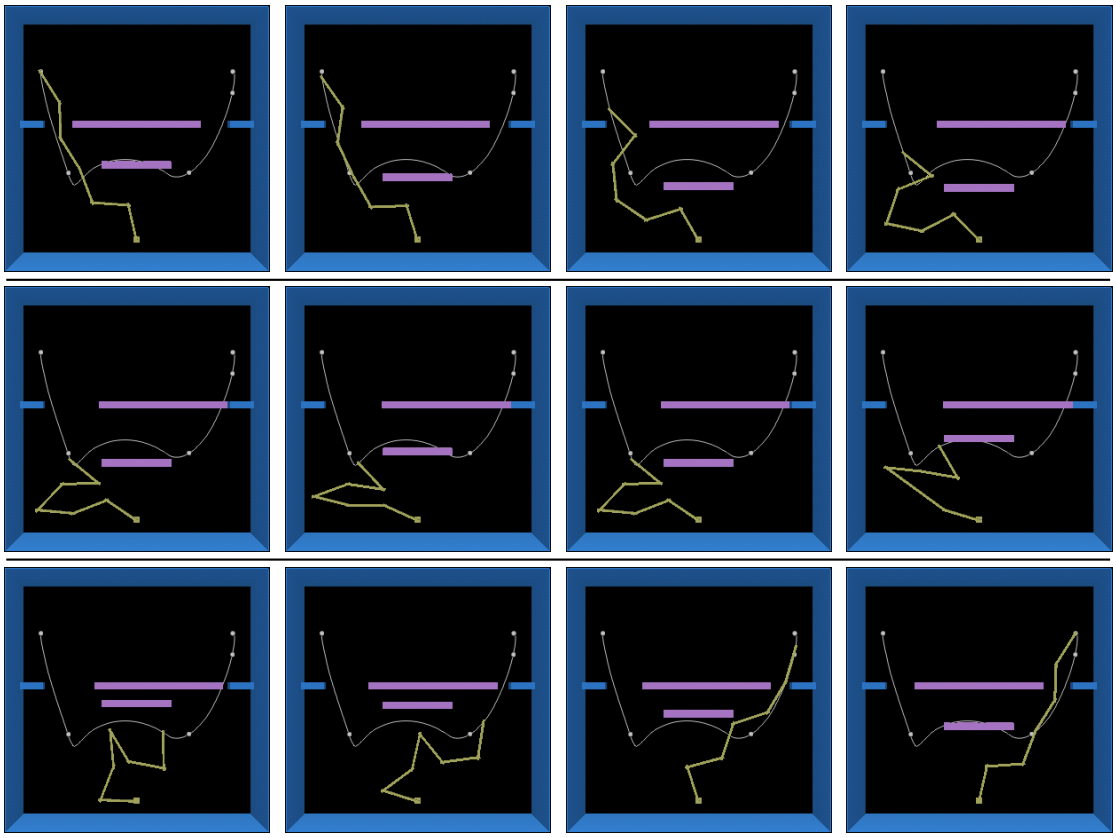

Figure 9.17: *Solved task with modified bang-bang path motion.*

10, a step size equal to 50, and an epsilon of 10^{-8}.

First, the robot waits until the lower bar has moved out of its way, before getting on the right side. At the end, the robot has to hurry to reach its goal position. This is only possible if the robot has enough torque to reach the goal after it has waited until the lower bar is out of its way.

For the next test, we kept the precision at 10 since this appears to be a good choice. With larger values the planner will not find its way though the gaps and smaller values result in a longer computing time. If the step size is decreased, then the computing time is increased (a step size of 20 results in a computation time of 115 seconds).

To reduce the capabilities of the robot, we changed the maximal torque to only 10 per cent of the original one (0.007 Nm). Under this circumstance, no result is found if the step size is too large. For a step size of 20, the computing time is 235 seconds. Now the robot's strategy changes a little since it cannot wait until the bar is out of its way. Instead, the robot tries to fold itself in such a way that it is small enough to pass the lower bar before the bar has started on its way upwards.

Finally, we want to analyse the scene using randomly chosen start and goal configurations. We let the torque be the original value again. For 1000 randomly chosen tasks with a starting time between 0 and 30 seconds and a goal time between 50 and 80 seconds, we obtained the following values for a step size of 20 and 50 (see Table 9.14).

step size	Count		Solved (%)		Time (sec)		Base Points	
20	1000	984	90	90	44.40	45.12	4.69	4.73
	904	888	100	100	36.64	37.30	4.61	4.66
	96		0		117.47		5.36	
50	1000	984	89	89	31.71	32.22	4.77	4.81
	887	871	100	100	25.54	26.01	4.69	4.74
	113		0		80.13		5.38	

Table 9.14: Results for sliding door environment with random tasks.

We note that a smaller step size solves more tasks, but also needs more computing time. Apparently it takes quite some time to reach the decision that the task cannot be solved. It would be interesting to know how many of the random tasks are actually solvable.

9.2.6 DLR Robot with Hand in Asteroid Field

In this last section we want to show that our planner is able to plan in high-dimensional search spaces with a complex environment consisting of a large number of time-varying obstacles.

The robot is a space robot developed for space travel at the DLR (German Aerospace Center) with a high-dimensional configuration space (Figure 9.18). The first three joints of the robot are prismatic joints such that the robot may move along the x, y, and z-axis. From there to the hand of the robot, seven rotational joints follow to give the robot liberty of action. The hand itself consists of four fingers. Each finger has four rotational joints, where two of them are located at the base of the finger. With this joint combination, the finger may move sideways or can be bent. The other two joints are for bending the finger such that an item can be grasped with the hand. In total, the robot has 26 joints.

In addition to the robot, there are seventeen small asteroids in this scene with various trajectories. The task for the space robot is to grasp the white object at a given time. In Figure 9.19, the initial trajectory of the task can be seen.

The robot has to avoid all the obstacles while grasping the white ball. In Figure 9.19, we have marked all collisions of the initial trajectory with white arrows. Figure 9.20 shows a possible solution for this task.

The task was solved in 30 minutes using a precision value of 10 and a step size of 20. The epsilon value was set to 0.001. The planning time is quite long, because the final position is very close to the obstacle that has to be grasped. Moreover, many collision tests are necessary since the time-varying obstacles are quite fast. The percentage of time used for the torque and force rating is about 6 per cent, for the collision test it is 39 per cent, and for the collision rating 41 per cent was used.

The start and goal configurations that we used in the above example are particularly challenging. To demonstrate this, we generated 100 random tasks for the same environment (with identical obstacle trajectories). In each of the new tasks, the starting time was set to 0 and the goal time was 60, while the rest of the start and goal configurations were

Figure 9.18: *The space robot with a high-dimensional configuration space.*

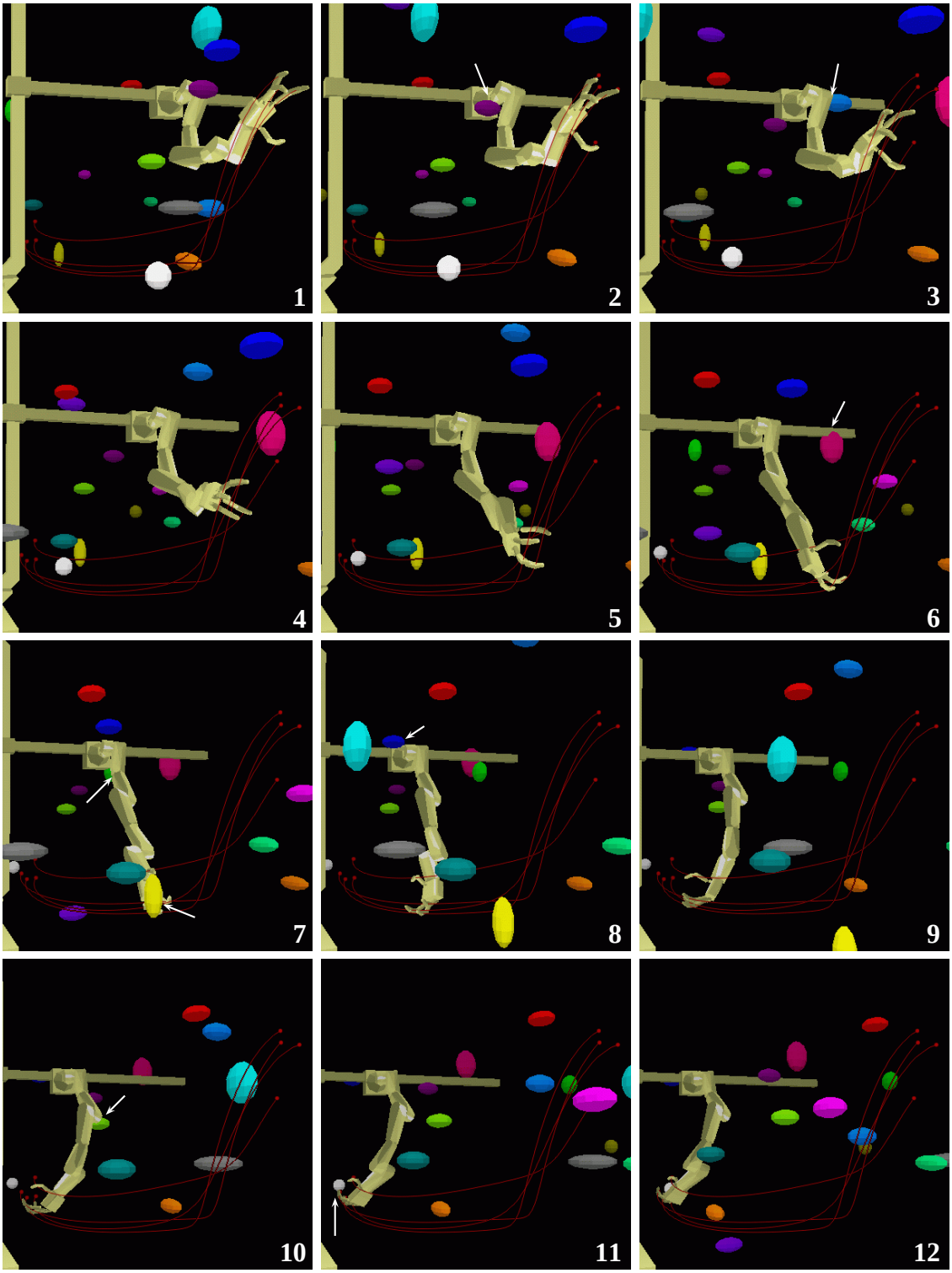

Figure 9.19: *Initial trajectory for the high-dimensional space robot.*

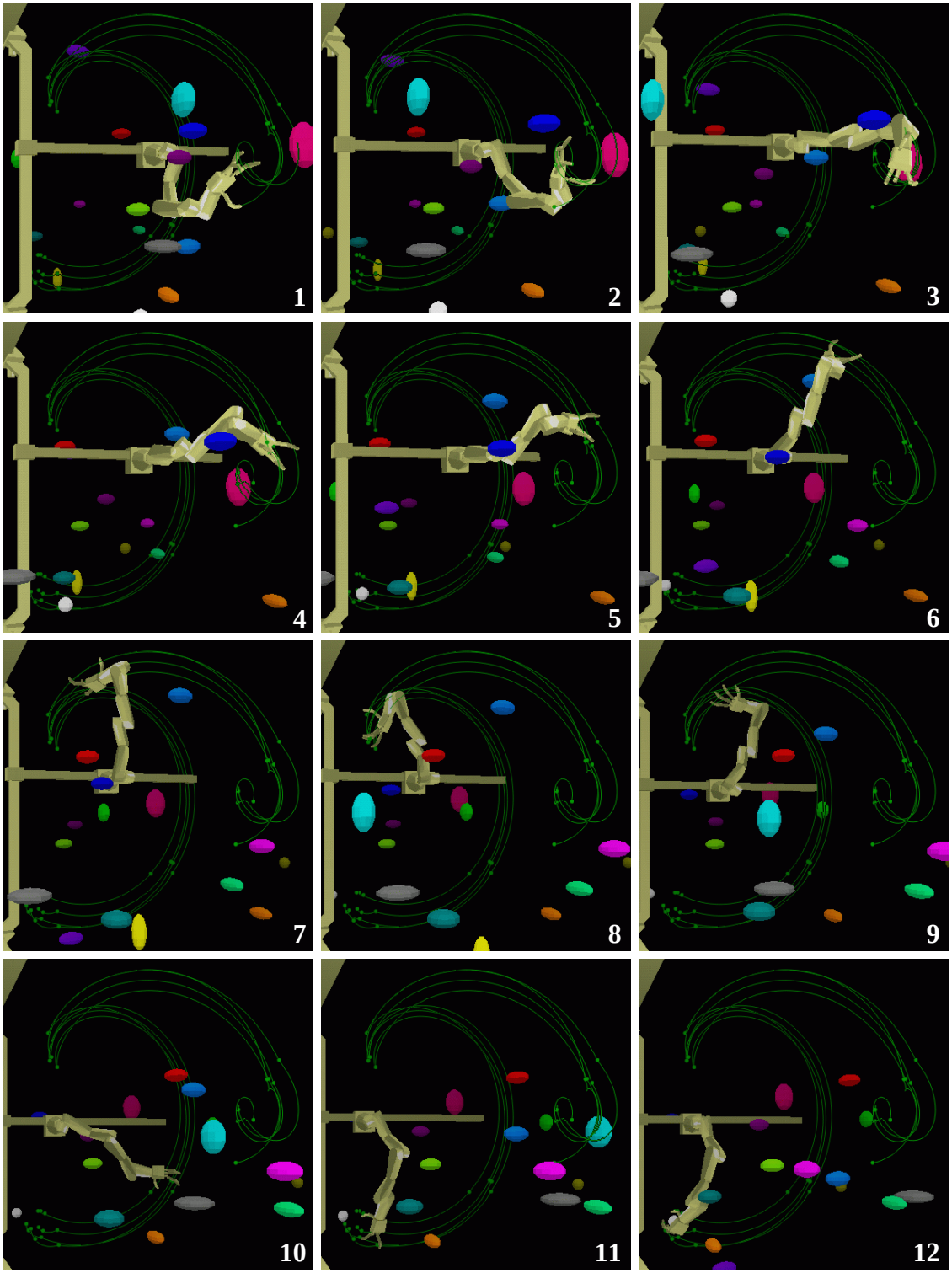

Figure 9.20: *Solved task for the high-dimensional space robot.*

chosen at random. The average running time of our planner on these tasks was 5.3 minutes (epsilon 0.001, step size and precision 20). None of the random tasks had an initial free trajectory. From these 100 tasks, 63 per cent were solved. Note that we do not know how many of the tasks are actually solvable. If we consider solved tasks only, then the average computing time was 4.98 min with 5.25 base points on average. For each unsolved task, the planner needed 5.85 min to signal failure and 5.19 base points were contained in the final trajectory on average.

9.3 Summary

We have shown in various experiments that our planning approach is successful in most planning scenarios. Our planner finds trajectories for robots with high-dimensional search spaces even in complex environments with narrow free space and time-varying obstacles. Currently, to the best of our knowledge, no other planning approach exists, which is capable of delivering reasonable planning results in such settings. A comparison with other planners is quite difficult since most planners do not consider time-varying environments or dynamic constraints of the robot. As far as benchmarks in static environments are concerned, we have compared our planner with the BB-method and the parallel A*-search algorithm. If the focus is on collisions only, there is no way to improve upon the BB-method. In comparison to the A*-search, our planner is not too bad, if we take into account, that we do our calculations only on a single processor.

CHAPTER 10

Conclusion

In this chapter we summarise our results and we take a look at promising directions for future research on motion planning in time-varying environments.

10.1 Summary

In this thesis, we have presented a motion planning algorithm for robots in time-varying environments. The algorithm finds collision-free trajectories that respect the dynamic constraints of the robot. Moreover, the algorithm is able to handle obstacles that move. To our knowledge, this algorithm is the first that plans motions in such a general setting.

The main concern of our research was to find a general, robust approach to motion planning that is capable of dealing with a high dimensional search space and a complex environment. The approach should be robust enough to be adaptable to specific needs and extendable to handle additional constraints. In our approach, three basic algorithmic components constantly interact to improve an initial trajectory until a trajectory is found that satisfies some given criteria. The three components are: a rating component that evaluates a given exact trajectory according to some given constraints, a search component that modifies a base point trajectory, and thirdly, a trajectory generation component that constructs an exact trajectory from a base point trajectory. Since the high dimensional search space prohibits its full exploration, our planner uses heuristics for a local search including randomisation to escape local minima. In contrast to the classic approach to motion planning, where path planning and trajectory generation are separate successive tasks, our approach considers both as a unit. At any time during the planning process, a geometric path together with a velocity profile for the robot's joints is manipulated and rated.

Each of the three components in our approach (rating, search, trajectory generation) is adaptable and extendable. Since the search component does not handle exact trajectories directly but only base points, the planning process can deal with different types of trajectories by simply adapting the trajectory generation component. It is even possible to use a different trajectory type for each joint of the robot. This might be necessary

in non-holonomic robots or robots with special types of joints. As trajectory types, we have analysed point-to-point motions and path motions. For each type we have tested polynomial connection between base points and a modified bang-bang connection with a trapezoid velocity profile. Two important criteria for selecting a motion type are smoothness and the locality of changes necessary if a single base point is moved. It turned out that in general the modified bang-bang path motion is a good compromise in that respect.

The rating component is responsible for determining the quality of a given exact trajectory. The interface between rating and searching is open enough to handle various kinds of rating criteria. Our focus in this thesis was on the mandatory constraints, namely freedom from collision and keeping the dynamic limits of the robot.

For collision rating we have adapted and extended an existing collision test. The collision test itself is based on the well-known oriented bounding box method. We have extended this method such that a trajectory cannot only be tested for freedom from collision with static obstacles but also with time-varying obstacles. Moreover, we have integrated a rating of the collision depth into the collision test. Such a rating is crucial in order to guide the planner out of a collision. The second rating function considers the violation of dynamic constraints of the robot. Here the quality of a trajectory is determined by how much the robot's force and torque limits are exceeded.

Other types of ratings that might be of interest reflect optimisation criteria such as total trajectory time or energy. We have shown how ratings of mandatory constraints and ratings of optimisation criteria can be integrated, taking into account that the requirements of the two rating types are quite different. For example, it does not make sense to optimise a trajectory for energy as long as it is still colliding.

To show the usefulness of our approach, we have implemented our planning algorithm within the scope of a robot simulation system and we have tested it in different scenarios. The results that we have obtained are quite promising. Our planner behaves well in most situations even in quite complex set-ups where the free configuration space is particularly narrow. A comparison of our results with other algorithms turned out to be difficult, since at the time being no other implementations for motion planning in time-varying environments for arbitrary manipulators exist (to our knowledge).

10.2 Future Research

We consider our work on motion planning in time-varying environments as a first step. There are still many open problems and there are several promising directions for future research on this topic. Examples are: planning for mobile robots, minimal time planning, and real-time motion planning.

A reasonable next step would be to add optimisation ratings to the overall trajectory rating. Minimal time and minimal energy would be particularly interesting. We have shown how these ratings could be included in our planning process. But there might be other ways of handling optimisation criteria. One might be tempted to find a minimal time trajectory by doing a bisection of the goal time, starting with a random goal time. However, this method is likely to fail in time-varying environments since increasing the goal time does not necessarily make it easier to find a trajectory. The goal configuration

might not even be free at the desired time.

Another interesting extension would be to integrate more mandatory constraints (besides freedom from collision and dynamic limits). Consider a robot that holds a cup filled with liquid. Here the constraint would be that the robot's tool is kept within a certain orientation interval relative to the world coordinate system. Or consider a robot that needs to keep its tool at an exact distance from some obstacle, for example, to weld or paint some object.

A quite interesting direction for further research is to develop more involved heuristics for moving a base point. One could switch between different strategies depending on which of the constraints is responsible for a bad rating. This way we might be able to decrease planning time, since we can use more specialised heuristics. For example, if a collision rating is bad we could do orthonormal movements of the links in the physical world while if torque and force rating is bad we move the base points in the time coordinate.

At the end of Chapter 5, we gave hints on how our approach can be used to plan trajectories for mobile robots. It would be interesting to explore this further, and to find suitable trajectory types, ratings, and dynamic models for non-holonomic robot types.

Another extension would be to analyse other more complex types of joint combinations, e.g. gripper, soft gripper, or helicoidal joints (as presented in Chapter 4). For these joint combinations appropriate dynamic models need to be found but we conjecture that our algorithm still works since the planning itself is done in the configuration space.

Let us now take a look at more involved problem settings. In our work we have dealt with the basic motion planning problem where a start and a goal configuration and two points in time are given. This setting could be varied in different ways. First, assume we are given a start configuration and a fixed starting time but the goal configuration is time-varying, that is, our planner can decide when the goal configuration is reached but depending on the time, the goal configuration changes. As for an example, consider a conveyor belt with moving workpieces. The robot's task is to grasp the next workpiece but the robot can decide by itself when to grasp it.

Another variant would be to have a goal configuration that is only partially specified in the problem instance. For example, to grasp a workpiece on a conveyer belt, it would suffice to specify that the robot's tool must have a certain position in the final configuration. The planner can decide itself how the final configuration looks as long as the tool is in the right position.

Another interesting problem extension is to add a number of intermediate robot configurations to the problem instance. The robot still needs to reach a goal configuration at a certain time but it also must reach each of the given intermediate configurations in the given order. The interesting part here is that the planner can decide at which points in time the intermediate configurations are reached.

In this work, we have considered motion planning in known environments, that is, all obstacles and their trajectories are known in advance. Sometimes, however, only partial information on the environment is available in the beginning and information about obstacles is gathered by sensors while the robot is moving. It might be worthwhile to evaluate how our algorithm can be adapted to planning in dynamic environments. One possible approach would be to start planning with the initial information, then to start the

robot movement and whenever the environment changes in such a way that the current trajectory is jeopardised, a new trajectory is planned and then executed. The above is iterated until the goal is reached. It would be necessary in each round to extrapolate the trajectories of the obstacles based on the sensor information gathered so far.

Finally, we would like to point out that our approach is well suited for parallelisation. First, since our rating is done piecewise for each pair of base points, the rating for the pieces may be done in parallel. And second, if a trajectory type is used where a modification of one base point only results in local changes to the exact trajectory, then a local search to improve the trajectory can be performed at several base points simultaneously.

List of Figures

List of Tables

List of Algorithms

In the following we first give a listing of the global parameters used in the planning process. Thereafter the functions of the planning algorithm are described. The page number after a function gives the first appearance of the function in the text. A more detailed description can be found there.

Parameter	**Description**
c_o	the planning process stops after the rating has reached c_o.
δ	δ determines the minimal time distance between two consecutive base points.
`down`	If `down` is `true`, then the modifications of base points is performed top-down along the robots' link topology (starting at the top joint). Otherwise the opposite order is used.
ε	ε is greater than 0 and determines whether a trajectory has a better rating than another trajectory.
`first`	If `first` is `true`, then the first modification is chosen that improves a rating. Otherwise the best of all base point movements is taken.
λ	λ determines the minimal step size that a base point is moved in one direction.
`max`	`max` is the total number of random movements that is allowed during the planning process.
p	p is the precision which is used to calculate the ratings of the exact trajectory. Higher values result in a less exact rating.
`random`	If `random` is `true`, then the bottom-up or top-down search in the base point dimensions starts at a randomly selected dimension.

Algorithm	Description
addBasePoint	The function *bool* addBasePoint (*trajectory* $\mathcal{T}_n$, *trajectory* $\mathcal{T}_o$, *basePoint b*) inserts the base point *b* into the trajectory $\mathcal{T}_o$. The resulting trajectory is written to $\mathcal{T}_n$. (see page 112)
addSection	The function *bool* addSection (*trajectory* $\mathcal{T}_n$, *trajectory* $\mathcal{T}_o$, *int* i) adds a new base point to $\mathcal{T}_o$ if the new trajectory $\mathcal{T}_n$ is better than the original one after moving. (see page 112)
deleteBasePoint	The function *bool* deleteBasePoint (*trajectory* $\mathcal{T}_n$, *trajectory* $\mathcal{T}_o$, *int i*) deletes the *i*-th base point. The resulting trajectory is written to $\mathcal{T}_n$. (see page 114)
deleteSection	The function *bool* deleteSection (*trajectory* $\mathcal{T}_n$, *trajectory* $\mathcal{T}_o$, *int* i) deletes the *i*-th base point in $\mathcal{T}_o$ if the new trajectory $\mathcal{T}_n$ is better than the original one after moving. (see page 113)
generateList	The function *list* generateList () generates a list depending on the search strategy (bottom-up, top-down, random). The elements in the list are the possible movements for a base point. (see page 109)
getMaxMovement	The function *double* getMaxMovement (*trajectory* $\mathcal{T}$, *double* t_s, *double* t_g) returns the maximal distance between any reference point on the robot (three for each link) at time t_s and t_g. (see page 72)
getSafetyDist	The function *double* getSafetyDist (*trajectory T*, *double* t_s, *double* t_g) returns the maximal safety distances needed to ensure freedom from collision in the time interval $[t_s, t_g]$. (see page 92)
getTestPosition	The function *double* getTestPosition (*trajectory T*, *double* t_s, *double* t_g) returns the position of the robot suitable for the collision test. (see page 93)
listFirst	The function *listElement* listFirst (*list l*) returns the first element of the list *l*. (see page 107)
listInsert	The function *list* listInsert (*list l*, *int j*) inserts the moving possibilities for dimension *j* at the front of list *l*. (see page 107)
listNext	The function *listElement* listNext (*listElement e*) returns the element after element *e*. (see page 107)
listReadTupel	The function *int* listReadTupel (*listElement e*, *int i*) reads the *i*-th element of tuple *e*. (see page 107)
listRemove	The function *list* listRemove (*list l*, *int j*) removes all elements of dimension *j* from the list and returns the new list. (see page 107)

Algorithm	Description
`maxRating`	The function *int* `maxRating` ($\mathcal{T}$) returns the index of the section with the worst rating in $\mathcal{T}$. (see page 109)
`move`	The function *bool* `move` (*trajectory* $\mathcal{T}_n$, *trajectory* $\mathcal{T}_o$, *int i*, *int j*, *double* d_p, *double* d_n) moves section *i* of the trajectory $\mathcal{T}_o$, by changing the *j*-th dimension of the configuration space. (see page 104)
`moveInterval`	The function *bool* `moveInterval` (*trajectory* $\mathcal{T}_n$, *trajectory* $\mathcal{T}_o$, *int* i, *int* j) tries to reach a better rating by modifying all sections from *i* to *j*. (see page 109)
`moveSection`	The function *bool* `moveSection` (*trajectory* $\mathcal{T}_n$, *trajectory* $\mathcal{T}_o$, *int* i) tries to improve the trajectory $\mathcal{T}_o$ by moving the section *i*. (see page 107)
`rate`	The function *rating* `rate` (*trajectory* $\mathcal{T}$, *int i*, *int j*) returns the rating of trajectory $\mathcal{T}$. (see page 101)
`rateCollision`	The function *double* `rateCollision` (*trajectory* $\mathcal{T}$, *double* t_s, *double* t_g) calculates the collision rating for the trajectory section between t_s and t_g. (see page 93)
`rateTorqueAndForce`	The function *double* `rateTorqueAndForce` (*trajectory* $\mathcal{T}$, *double* t_s, *double* t_g) calculates the force and torque rating for the trajectory section between t_s and t_g. (see page 73)
`rateTorqueAnd-` `-ForceEmbedded`	The function *void* `rateTorqueAndForceEmbedded` (*trajectory* $\mathcal{T}$, *double* t_s, *double* t_g, *double*[] τ) is used by the function `rateTorqueAndForce`. (see page 73)
`smaller`	The function *bool* `smaller` (*trajectory* $\mathcal{T}_1$, *trajectory* $\mathcal{T}_2$) compares two trajectories and returns true if $\mathcal{T}_1$ is epsilon smaller ($<^{\varepsilon}$) than $\mathcal{T}_2$. (see page 102)
`zero`	The function *bool* `zero` (*trajectory* $\mathcal{T}$) checks whether all section ratings of the trajectory $\mathcal{T}$ are smaller or equal to the global parameter c_o. (see page 102)

List of Symbols

What follows is a list of symbols together with short descriptions. If there is a definition or a detailed description for the symbol in this work, then the page number is given as well. By convention, A, B, and C are boolean matrices. Vectors are denoted by Greek or small letters with an arrow on top ($\vec{\omega}$). The i-th value of some vector $\vec{\omega}$ is denoted by ω_i.

Symbol	Description
$\circ$	The scalar product between two vectors is $\vec{v} \circ \vec{w} = \lvert\vec{v}\rvert \ \lvert\vec{w}\rvert \ \cos\phi$, where ϕ is the angle between the two vectors.
$\lvert\vec{v}\rvert$	The length of a vector $\vec{v}$ is the square root of the sum over all squares of the components $\lvert\vec{v}\rvert = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$.
$A \wedge B$	The Boolean product of the Boolean matrices A and B. (see Definition 4.2 on Page 29)
B^i	The Boolean power of the Boolean matrix B. (see Definition 4.3 on page 29)
$A \vee B$	The Boolean union of the Boolean matrices A and B. (see Definition 4.4 on page 29)
B^*	The transitive closure of the Boolean matrix B. (see Definition 4.5 on page 30)
$\hat{B}^*$	The modified transitive closure of the Boolean matrix B representing the topology of the robot and the dependency functions. (see Definition 4.11 on page 35)
λ_j	The type of joint J_j is determined by λ_j. (see Equation 4.5 on page 31)
$\vec{\upsilon}$	The joint vector of the robot, representing the joint positions. (see Definition 4.7 on page 33)
υ_j	The value of the j-th joint of the robot. (see Definition 4.7 on page 33)
$\upsilon_j^{\min}$, $\upsilon_j^{\max}$	The minimal or maximal value the j-th joint of the robot may have. (see Definition 4.7 on page 33)

Symbol	Description
$\vec{\upsilon}_d$	The dynamic joint configuration $\vec{\upsilon}_d = (\vec{\upsilon}_\omega, \vec{\upsilon}_\sigma, \vec{\upsilon}_\alpha)$ of the robot. (see Definition 4.16 on page 41)
$\vec{\upsilon}_\omega$	The position part of the robot's dynamic configuration $\vec{\upsilon}$; often we drop the ω and write only $\vec{\upsilon}$. (see Definition 4.16 on page 41)
$\vec{\upsilon}_\sigma$	The velocity part of the robot's dynamic configuration $\vec{\upsilon}$. (see Definition 4.16 on page 41)
$\vec{\upsilon}_\alpha$	The acceleration part of the robot's dynamic configuration $\vec{\upsilon}$. (see Definition 4.16 on page 41)
$\vec{\tau}$	The current forces and torques in the joints of the robot. (see Chapter 4.4 on page 39)
$\vec{\vartheta}$	The configuration vector of the robot extended by the time. (see Definition 4.13 on page 38)
$\vec{\sigma}_{O_i}$	The maximal speed vector of obstacle O_i. (see Chapter 4.2 on page 36)
$\theta_j^{\max}$	The maximal torque or force the j-th joint of the robot has. (see Chapter 4.1.1 on page 30)
$\vec{\omega}$	The configuration vector of the robot. (see Definition 4.7 on page 33)
ω_i	The value of dimension i of the configuration. (see Definition 4.7 on page 33)
$\vec{\omega}_i$	The configuration vector of the i-th base point. (see Chapter 5 on page 45)
B	The boolean $(n+1) \times (n+1)$ matrix describing the topology of the robot. (page 27)
$\vec{b}_i$	The i-th base point $\vec{b}_i = (\vec{\omega}_i, t_i)$ of a base point trajectory. (see Chapter 5 on page 45)
$c_{\vec{o}}^a$	The collision value for the triangle a in the environment based on the orientation vector $\vec{o}$ (page 83).
C_l	The collision value for the link L_l in the environment based on the orientation vector $\vec{o}_l$ of the link (page 84).
$C(\vec{\upsilon}, t, d)$	The collision value for the robot in position $\vec{\upsilon}$ at time t with safety distance d. (page 84).
$\mathcal{CS}$	Configuration space containing all valid configurations. (see Definition 4.7 on page 33)
$\mathcal{CS}_c$	Configuration space containing all colliding time configurations. (see Chapter 4.3 on page 38)
$\mathcal{CS}_f$	Configuration space containing all free time configurations. (see Chapter 4.3 on page 38)
$\mathcal{CS}_t$	Configuration space containing all valid configurations extended by the dimension of the time. (see Definition 4.13 on page 38)
$d(j)$	Index dependency function returning the dimension of joint J_j in the configuration space. (see Definition 4.6 on page 33)

Symbol Description

${}^0D_j$	The homogeneous transformation matrix representing the location of joint J_j in the world. (see Chapter 4.1.1 on page 30)
${}^iD_j$	The homogeneous transformation matrix from the coordinate system of joint J_j to joint J_i. (see Chapter 4.1.1 on page 30)
D_{O_i}	The homogeneous transformation matrix describing the location of the coordinate system of obstacle O_i. (see Chapter 4.2 on page 36)
${}^jE$	The homogeneous transformation matrix of joint J_j from the joint's coordinate system in zero position to the actual position. (page 30)
$f_j(x)$	Value dependency function returning the value of joint J_j if the value in the configuration space $d(j)$ is x. (see Definition 4.6 on page 33)
g_i	Position of centre of gravity of link L_i relative to the coordinate system of joint J_i. (see Chapter 4.1.2 on page 35)
I_i	The 3×3 inertia tensor matrix of link L_i relative to the centre of gravity of link L_i. (see Chapter 4.1.2 on page 35)
$\mathcal{J}$	Set of all joints $J_1, \ldots, J_n$ of the robot. (see Chapter 4.1 on page 27 and Chapter 4.1.1 on page 30)
$\mathcal{J}_j^{\mathbf{pj}}$	Set of previous joints of joint J_j. (see Definition 4.8 on page 34)
$\mathcal{J}_j^{\mathbf{nj}}$	Set of next joints of joint J_j. (see Definition 4.9 on page 34)
$\mathcal{J}_j^{\mathbf{nl}}$	Set of next links of joint J_j. (see Definition 4.10 on page 34)
J_i	The i-th joint in the joint set $\mathcal{J}$. (see Chapter 4.1 on page 27 and Chapter 4.1.1 on page 30)
$\mathcal{L}$	Set of all links $L_0, \ldots, L_n$ of the robot. (see Chapter 4.1 on page 27 and Chapter 4.1.2 on page 35)
$\mathcal{L}_j^{\mathbf{pj}}$	Set of previous joints of link L_j. (see Definition 4.12 on page 35)
L_i	The i-th link in the link set $\mathcal{L}$. (see Chapter 4.1 on page 27 and Chapter 4.1.2 on page 35)
M	M is a function $\mathbb{R}^{3n} \to \mathbb{R}^n$ witch takes a dynamic configuration $\vec{\upsilon}_d$ and returns the force or torque values $\vec{\tau}$ for each joint. (see Chapter 4.4 on page 39)
m_i	The mass of link L_i. (see Chapter 4.1.2 on page 35)
O	Set of all obstacles $O_1, \ldots, O_o$ of the environment. (see Chapter 4.2 on page 36)
O_i	The i-th obstacle in the environment. (see Chapter 4.2 on page 36)
$o_i(t)$	The time-varying location of obstacle i. (see Chapter 4.2 on page 36)
$p(j)$	Function returning the index of the previous link or the previous joint of the link L_j or the joint J_j respectively. (see Equation 4.3 on page 30)
${}^0p_j$	The position of the coordinate system of joint J_j in the world. (see Chapter 4.1.1 on page 30)
${}^ip_j$	The position of the coordinate system of joint J_j relative to joint J_i. (see Chapter 4.1.1 on page 30)

Symbol	Description
${}^0R_j$	The 3×3 rotation matrix part representing the rotation of joint J_j in the world. (see Chapter 4.1.1 on page 30)
${}^iR_j$	The 3×3 rotation matrix from the coordinate system of joint J_j to joint J_i. (see Chapter 4.1.1 on page 30)
$\mathcal{R}$	Is a tuple of three elements $(\mathcal{J}, \mathcal{L}, B)$. The first is the set of joint $\mathcal{J}$, the second is the set of links $\mathcal{L}$, and the third is the boolean $(n+1) \times (n+1)$ matrix B describing the topology of the robot. (see Chapter 4.1 on page 27)
S_i	Trajectory section between $\vec{b}_{i-1}$ and $\vec{b}_i$. (page 45)
$\mathcal{T}_O$	The hierarchical representation of all triangles of an object O in an oriented bounding box tree (page 79).)
$T(t)$	Function of the time describing the trajectory in the configuration space dimension. (see Chapter 4.4 on page 39)
$T^D(t)$	Function of the time describing the trajectory in the real robot joints returning the position, velocity, and acceleration in the joints. (see Definition 4.16 on page 41)
$T_i(t)$	Function of the time describing the trajectory in the i-th dimension of the configuration space dimension. (see Chapter 4.4 on page 39)
$\dot{T}(t)$	Function of the time describing the first derivations of the trajectory in the configuration space dimension. (see Chapter 4.4 on page 39)
$\dot{T}_i(t)$	Function of the time describing the first derivation of the trajectory in the i-th dimension of the configuration space dimension. (see Chapter 4.4 on page 39)
$\ddot{T}(t)$	Function of the time describing the second derivations of the trajectory in the configuration space dimension. (see Chapter 4.4 on page 39)
$\ddot{T}_i(t)$	Function of the time describing the second derivation of the trajectory in the i-th dimension of the configuration space dimension. (see Chapter 4.4 on page 39)

Bibliography

Boris Baginski. Local motion planning for manipulators based on shrinking and growing geometry models. In *Proceedings of the 1996 IEEE International Conference on Robotics and Automation (ICRA'96)*, pages 3303–3308, Minneapolis, Minnesota, April 1996a.

Boris Baginski. The Z^3-method for fast path planning in dynamic environments. In M.H. Hamza, editor, *Proceedings of the IASTED Conference on Applications of Control and Robotics*, pages 47–52, Orlando, Florida, January 1996b. IASTED Acta Press, Anaheim, Calgary, Zürich.

Boris Baginski. Efficient dynamic collision detection using expanded geometry models. In *Proceedings of the IEEE/RSJ/GI International Conference on Intelligent Robots and Systems (IROS'97)*, pages 1714–1719, Grenoble, France, September 1997.

Boris Baginski. *Motion Planning for Manipulators with Many Degrees of Freedom - The BB-Method*. Number 195 in DISKI (Dissertationen zur Künstlichen Intelligenz). Infix-Verlag, Sankt Augustin, 1999.

Gill Barequet, Matthew Dickerson, and David Eppstein. On triangulating three-dimensional polygons. *Computational Geometry: Theory and Applications (CGTA)*, 10(3):155–170, June 1998.

Jérôme Barraquand, Lydia Kavraki, Jean-Claude Latombe, Tsai-Yen Li, Rajeev Motwani, and Prabhavan Raghavan. A random sampling scheme for path planning. *International Journal of Robotics Research*, 16(6):759–774, 1997.

Jérôme Barraquand, Bruno Langlois, and Jean-Claude Latombe. Robot motion planning with many degrees of freedom and dynamic constraints. In H. Miura and S. Arimoto, editors, *Robotics Research*, volume 5, pages 435–444. MIT Press, 1990.

Jérôme Barraquand, Bruno Langlois, and Jean-Claude Latombe. Numerical potential field techniques for robot path planning. *IEEE Transactions on Systems, Man and Cybernetics*, 22(2):224–241, March 1992.

Jérôme Barraquand and Jean-Claude Latombe. A Monte-Carlo algorithm for path planning with many degrees of freedom. In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation (ICRA'90)*, pages 1712–1717, Cincinnati, Ohio, May 1990.

Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient robust access method for points and rectangles. In Hector Garcia-Molina and H.V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, NJ, May 1990.

James E. Bobrow, Steven Dubowsky, and J.S. Gibson. Time-optimal control of robotic manipulators along specified paths. *International Journal of Robotics Research*, 4(3): 3–17, 1985.

Rodney A. Brooks. Solving the find-path problem by good representation of free space. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(3):190–197, March 1983.

Rodney A. Brooks and Tomás Lozano-Pérez. A subdivision algorithm in configuration space for findpath with rotation. *IEEE Transactions on Systems, Man, and Cybernetics*, 15:224–233, 1985.

Charles E. Buckley. A foundation for the "flexible–trajectory" approach to numeric path planning. *International Journal of Robotics Research, MIT Press*, 8(3):44–64, June 1989.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, Chichester, New York, Brisbane, Toronto, Singapore, 1996.

Maik Buttelmann, Martin Kieren, and Boris Lohmann. Trajektoriengenerierung und Bahnregelung für nichtholonome, autonome Fahrzeuge. In *Autonome Mobile Systeme 1999*, volume 15 of *Informatik aktuell*, pages 303–312. Günther Schmidt and Uwe Hanebeck and Franz Freyberger, November 1999.

John F. Canny. *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, Massachusetts, 1988.

Daniel Challou, Daniel Boley, Maria Gini, Vipin Kumar, and Curtis Olson. Parallel search algorithms for robot motion planning. In Kamal K. Gupta and Angel P. del Pobil, editors, *Practical Motion Planning in Robotics: Current Approaches and Future Directions*, pages 115–131. John Wiley and Sons, 1998.

Pang C. Chen and Yong K. Hwang. SANDROS: A motion planner with performance proportional to task difficulty. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation (ICRA'92)*, pages 2346–2353, Nice, France, May 1992.

Pang C. Chen and Yong K. Hwang. Performance of the SANDROS planner. Technical report, Sandia National Laboratories, Albuquerque, New Mexico, April 1996.

James J. Craig. *Introduction to Robotics. Mechanics and Control*. Addison-Wesley, 1986.

Jacques Denavit and Richard S. Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. *ASME Journal of Applied Mechanics*, 22(77):215–221, June 1955.

Bruce R. Donald, Patrick Xavier, John Canny, and John Reif. Kinodynamic motion planning. *Journal of the ACM*, 40(5):1048–1066, November 1993.

Bruce R. Donald and Patrick G. Xavier. Provably good approximation algorithms for optimal kinodynamic planning for cartesian robots and open chain manipulators. *Algorithmica*, 14(6):480–530, December 1995a.

Bruce R. Donald and Patrick G. Xavier. Provably good approximation algorithms for optimal kinodynamic planning: Robots with decoupled dynamics bounds. *Algorithmica*, 14(6):443–479, December 1995b.

G. Duelen and C. Willnow. Path planning of transfer motions for industrial robots by heuristically controlled decomposition of the configuration space. In *Proceedings of the European Robotics and Intelligent System Conference EURISCON'91*, pages 217–224, Corfu, Greece, June 1991.

Martin Eldracher. *Planung kinematischer Trajektorien für Manipulatoren mit Hilfe von Subzielen und neuronalen Netzen*. Number 111 in DISKI (Dissertationen zur Künstlichen Intelligenz). Infix-Verlag, Sankt-Augustin, 1996.

Martin Eldracher and Peter Baumann. Kinematic path planning for manipulators in dynamic environments using adaptive neural models and neural subgoal generation. Technical report, Technische Universität München, Munich, Germany, 1995.

Michael Erdmann and Tomás Lozano-Pérez. On multiple moving objects. *Algorithmica*, 2, 1987.

Paolo Fiorini. *Robot Motion Planning Among Moving Obstacles*. PhD thesis, University of California, Los Angeles, CA, 1995.

Paolo Fiorini and Zvi Shiller. Motion planning in dynamic environments using velocity obstacles. *The International Journal of Robotics Research*, 17(7):760–772, July 1998.

Steve Fortune. Voronoi diagrams and delaunay triangulations. In D.-Z. Du and F. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 193–233. World Scientific Publishers, 1992.

Bernhard Glavina. *Planung kollisionsfreier Bewegungen für Manipulatoren durch Kombination von zielgerichteter Suche und zufallsgesteuerter Zwischenzielerzeugung*. PhD thesis, Technische Universität München, February 1991.

Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics: Proceeding ACM SIGGRAPH*, pages 171–180, 1996.

Kamal K. Gupta. Overview and state of the art. In Kamal K. Gupta and Angel P. del Pobil, editors, *Practical Motion Planning in Robotics: Current Approaches and Future Directions*, pages 3–8. John Wiley and Sons, 1998.

Dan Halperin, Lydia Kavraki, and Jean-Claude Latombe. Robotics. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 755–778. CRC Press, Boca Raton New York, July 1997.

Greg Heinzinger, Paul Jacobs, John Canny, and Bred Paden. Time-optimal trajectories for a robot manipulator: A provably good approximation algorithm. In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation (ICRA'90)*, pages 150–156, 1990.

Domenik Henrich, Christian Wurll, and Heinz Wörn. 6 dof path planning in dynamic environments - a parallel on-line approach. In *Proceedings of the 1998 IEEE International Conference on Robotics and Automation (ICRA'98)*, pages 330–335, Leuven, Belgium, May 1998.

Dominik Henrich and Xiaoqing Cheng. Fast distance computation for on-line collision detection with multi-arm robots. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation (ICRA'92)*, pages 2514–2519, Nice, France, May 1992.

Shigeo Hirose. *Biologically inspired robots: Snake-like Locomotors and Manipulators*. Oxford Science Publications, 1993.

Gerd Hirzinger, Jörg Butterfass, Markus Grebenstein, Matthias Hähnle, Ingo Schäfer, and Norbert Sporer. Robonauts need light-weight arms and articulated hands. In *Proceedings of the 6th ESA Workshop on ASTRA*, Noordwijk, Netherlands, 2000.

Christoph M. Hoffmann. Solid modeling. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 863–880. CRC Press, Boca Raton New York, July 1997.

John E. Hopcroft, Deborah A. Joseph, and Sue H. Whitesides. On the movement of the robot arms in 2-dimensional bounded regions. *SIAM Journal on Computing*, 14(2): 315–333, 1985.

John E. Hopcroft, Jacob T. Schwartz, and Micha Sharir. Efficient detection of intersections among spheres. *International Journal of Robotics Research*, 2(4):77–80, 1983.

John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.

Manfred Husty, Adolf Karger, Hans Sachs, and Waldemar Steinhilper. *Kinematik und Robotik*. Springer-Verlag Berlin Heidelberg New York, 1997.

Yong K. Hwang. Completeness vs. efficiency in real applications of motion planning. In Kamal Gupta and Angel P. del Pobil, editors, *Practical Motion Planning in Robotics, Workshop WT2 at the IEEE International Conference on Robotics and Automation*, Minneapolis, Minnesota, April 1996.

Yong K. Hwang and N. Ahuja. Gross motion planning – A survey. *ACM Computing Surveys*, 24(3):219–291, September 1992.

Paul Jacobs, Greg Heinzinger, John Canny, and Brad Paden. Planning guaranteed near-time-optimal trajectories for a manipulator in a cluttered workspace. In *Proceeding International Workshop on Sensorial Integration for Industrial Robots: Architectures and Applications*, pages 162–167, November 1989.

K. Kant and Steve W. Zucker. Towards efficient trajectory planning: the path-velocity decomposition. *The International Journal of Robotic Research*, 5(3):72–89, 1986.

K. Kant and Steve W. Zucker. Planning collision free trajectories in time-varying environments: a two level hierarchy. In *Proceedings of the 1988 IEEE International Conference on Robotics and Automation (ICRA'88)*, pages 1644–1649, Raleigh, NC, 1988.

Lydia Kavraki, Mihail N. Kolountzakis, and Jean-Claude Latombe. Analysis of probabilistic networks for path planning. In *Proceedings of the 1996 IEEE International Conference on Robotics and Automation*, pages 3020–3025, Minneapolis, MN, 1996a.

Lydia Kavraki and Jean-Claude Latombe. Randomized preprocessing in configuration space for fast path planning. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation (ICRA'94)*, pages 2138–2145, 1994a.

Lydia Kavraki and Jean-Claude Latombe. Randomized preprocessing in configuration space for fast path planning: Articulated robots. In *Proceedings of the 1994 IEEE/RSJ/GI International Conference on Intelligent Robots and Systems (IROS'94)*, pages 1764–1771, Munich, Germany, 1994b.

Lydia Kavraki, Petr Svestka, Jean-Claude Latombe, and Mark Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996b.

Otto Kerne, Joseph Maurer, Jutta Steffens, Thomas Thode, and Rudolf Voller. *Vieweg Mathematik Lexikon*. Friedrich Vieweg & Sohn, Braunschweig Wiesbaden, 1988.

Robert Kindel, David Hsu, Jean-Claude Latombe, and Stephen Rock. Kinodynamic motion planning amidst moving obstacles. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation (ICRA'00)*, 2000.

Jean-Claude Latombe. *Robot Motion planning*. Kluwer Academic Publishers, 4th edition, 1996.

C. Laugier and F. Germain. An adaptive collision-free trajectory planner. In *Proceedings of the International Conference on Advanced Robotics*, Tokyo, Japan, 1985.

B.H. Lee and Y.P. Chien. Time-varying obstacle avoidance for robot manipulators: Approaches and difficulties. In *Proceedings of the IEEE 1987 International Conference on Robotics and Automation (ICRA'87)*, 1987.

Tomás Lozano-Pérez. Automatic planning of manipulator transfer movements. *IEEE Transactions Systems, Man and Cybernetics*, SMC-11:681–689, 1981.

Tomás Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, 32:108–120, 1983.

Johnson Y.S. Luh and Chyuan S. Lin. Optimal path planning for mechanical manipulators. *ASME Journal of Dynamic Systems, Measurement and Control*, pages 142–151, 1981.

Johnson Y.S. Luh and Michael W. Walker. Minimum-time along the path for a mechanical arm. In *Proceedings 1977 IEEE Conference Dec. Contr*, volume 2, pages 755–759, 1977.

Begoña Martínez-Savador, Angel P. del Pobil, and Miguel Pérez-Francisco. Very fast collision detection for practical motion planning part i: The spatial representation. In *Proceedings of the 1998 IEEE International Conference on Robotics and Automation (ICRA'98)*, pages 624–629, Leuven, Belgium, May 1998.

Brain Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California, December 1996.

Akira Mohri, Xiang D. Yang, and Motoji Yamamoto. Collision free trajectory planning for manipulators using potential function. In *Proceedings of the 1995 IEEE International Conference on Robotics and Automation (ICRA'95)*, pages 3069–3074, 1995.

J.J. Murray and C.P. Neumann. Linearization and sensitivity models of the Newton-Euler dynamic robot model. *ASME Journal of Dynamic Systems, Measurement, and Control*, 108:272–276, September 1986.

Nils J. Nilsson. A mobile automaton: An application of artificial intelligence techniques. In D.E. Walker and L.M. Norton, editors, *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 509–520, Washington, D.C., May 1969. W. Kaufmann.

Colm Ó'Dúnlaing and Chee K. Yap. A retraction method for planning the motion of a disc. *Journal of Algorithms*, 6, 1982.

J.K. Ousterhout. *Tcl und Tk: Entwicklung grafischer Benutzerschnittstellen für das X Windows System*. Addision-Wesley, Deutschland, 1995.

Hiroaki Ozaki and Chang-jun Lin. Optimal b-spline joint trajectory generation for collision-free movements of a manipulator under dynamic constraints. In *Proceedings of the 1996 IEEE International Conference on Robotics and Automation (ICRA'96)*, pages 3592–3597, Minneapolis, Minnesota, April 1996.

Hongjun Pu. *Novel algorithms towards systematic, generalized and compact recursive generation and computation of the complete robot dynamics with realisation aspects in CAD-systems*. Number 687 in Verein Deutscher Ingenieure: Fortschrittberichte VDI. VDI-Verlag, Düsseldorf, 1998.

Caigong Qin and Dominik Henrich. Path planning for industrial robot arms - a parallel randomized approach. In *Proceedings of the International Symposium on Intelligent Robotic Systems (SIRS'96)*, pages 65–72, July 1996.

Sean Quinlan. Efficient distance computation between non-convex objects. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation (ICRA'94)*, volume 4, pages 3324–3329, San Diego, California, May 1994.

John Reif and Micha Sharir. Motion planning in the presence of moving obstacles. In *Proceedings of the 26th IEEE Symposium on Foundations of Computer Science*, pages 144–154, Portland, OR, October 1985.

John Reif and Micha Sharir. Motion planning in the presence of moving obstacles. *Journal of the ACM*, 41(4):764–790, July 1994.

John H. Reif. Complexity of the mover's problem and generalizations (extended abstract). In *20th Annual Symposium on Foundations of Computer Science*, pages 421–427, San Juan, Puerto Rico, October 1979.

Elon Rimon and Daniel E. Koditschek. Exact robot navigation using artifical potential functions. *IEEE Transactions on Robotics and Automation*, 8(5):501–518, October 1992.

Hanan Samet. Hierarchical spatial data structures. In Y.-F. Wang, editor, *Design and Implementation of Large Spatial Databases*, volume 409 of *Lecture Notes in Computer Science*, pages 193–212, Santa Barbara, California, July 1989. Springer.

Jacob T. Schwartz and Micha Sharir. On the "piano movers" problem I. the case of a two-dimensional rigid polygonal body moving amidst polygonal barriers. *Communication on Pure Applied Mathematics*, 36:345–398, 1983a.

Jacob T. Schwartz and Micha Sharir. On the piano movers problem II. general techniques for computing topological properties of real algebraic manifolds. *Advances in applied mathematics*, 4:298–351, 1983b.

Micha Sharir. Algorithmic motion planning in robotics. *Computer*, 22(3):9–20, 1989.

Zvi Shiller, Hai Chang, and Vincent Wong. The practical implementation of time-optimal control for robotic manipulators. *Robotics & Computer-Integrated Manufacturing*, 12 (1):29–39, 1996.

Zvi Shiller and Steven Dubowsky. On computing the global time-optimal motions of robotic manipulators in the presence of obstacles. *IEEE Transactions on Robotics and Automation*, 7(6):785–797, December 1991.

Zvi Shiller, Frederic Large, and Sepanta Sekhavat. Motion planning in dynamic environments: Obstacles moving along arbitrary trajectories. In *Proceedings of the 2001 IEEE International Conference on Robotics and Automation (ICRA'01)*, pages 3716–3721, Seoul, Korea, May 2001.

K.G. Shin and N.D. McKay. Open-loop minimum-time control of mechanical manipulators and its application. In *Proceedings of the 1984 American Control Conference*, pages 1231–1236, 1984.

Hans-Jürgen Siegert and Siegfried Bocionek. *Robotik: Programmierung intelligenter Roboter*. Springer-Verlag Berlin Heidelberg New York, 1996.

Sabine Stifter. Mathematische Verfahren für Kollisionsprobleme bei Robotern. In Chatterji-Kulisch-Langwitz-Liedl-Purkert, editor, *Jahrbuch Überblicke Mathematik*, pages 61–76. Vieweg, 1991.

H. Weghorst, G. Hopper, and D. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, pages 52–69, 1984.

Maison Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*. Addision-Wesley, 1997.

Heinz Wörn, Dominik Henrich, and Christian Wurll. Motion planning in dynamic environments - a parallel online approach. In *3rd International Symposium on Artificial Life and Robotics (AROB'98)*, Oita, Japan, January 1998.

Motoji Yamamoto, Yukihiro Isshiki, and Akira Mohri. Collision free minimum time trajectory planning for manipulators using global search and gradient method. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'94)*, pages 2184–2191, 1994.

Jianwei Zhang. *Ein integriertes Verfahren zur effizienten Planung und Ausführung von Roboterbewegungen in unscharfen Umgebungen*. Number 77 in DISKI (Dissertationen zur Künstlichen Intelligenz. Infix-Verlag, Sankt Augustin, 1995.

David Zhu and Jean-Claude Latombe. New heuristic algorithms for efficient hierarchical path planning. *IEEE Transactions on Robotics and Automation*, 7(1):9–20, February 1991.