Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Section Algorithms and Visualization

Dynamic real-time collision detection for packaging robots

Master's Thesis

Ronald van Zon

Supervisors:
dr. Kevin Buchin (TU/e)
dr. Rodrigo Ignacio Silveira (UPC)
Igor Jovanovic B.Sc. (Omron)
Diego Escudero M.Sc. (Omron)

Eindhoven, August 2014

Abstract

Due to the increasing demand in higher performance (throughput and efficiency) of robotics packaging systems, while keeping the system's footprint as small as possible, robots must operate closer to each other. This raises the problems of collision detection and collision avoidance. In this work we focus on designing algorithms for collision detection between multiple (parallel) Delta robots, which are widely used in the packaging industry. The algorithms must operate in a real-time controller, which puts additional constraints on the processing power and memory requirements. We discuss several algorithmic approaches to this problem.

Acknowledgements

Without the help of my (theoretical) supervisors Kevin Buchin and Rodrigo Silveira, I could not have completed this thesis. Therefore, I would like to thank them for their support, ideas and guidance. I am also grateful to Mark de Berg, who provided me with new insights and ideas. I would like to thank Rodrigo, Diego Escudero and Renzo Slijp for their extensive feedback on my thesis.

Further, I would like to thank my (operational and practical) supervisors Igor Jovanovic and Diego Escudero for their help during my project. I thank Igor for guiding me regarding operational aspects and to find my place in the team in Omron, Barcelona. I thank Diego and Raffaele Vito for their extensive help during all (practical) problems I faced during the project. Further, I would like to thank all other team members, Santiago Arango, David Arcas, Ferran Carlas and Shinichi Hosomi for their help, support and friendliness during my stay in Barcelona.

I am grateful to Dan Halperin, Oren Salzman and Kiril Solovey, for their support, ideas and discussions. Most of the ideas raised have found their way to this thesis (in one way or another).

Finally, I would like to thank my family and friends, who supported me during my studies.

Contents

Contents vii

List of Figures ix

List of Tables xi

1 Introduction 1
- 1.1 Packaging robots . . . 2
- 1.2 Collision detection . . . 5
- 1.3 Real-time controller . . . 7
- 1.4 Problem statement . . . 7
- 1.5 Related work . . . 8
- 1.6 Results and organization . . . 9

2 Robot mechanics and model 11
- 2.1 Terminology . . . 11
- 2.2 Robot mechanics . . . 11
- 2.3 Coordinate systems . . . 13
- 2.4 Geometric modeling and transformations . . . 15
  - 2.4.1 Geometric modeling . . . 15
  - 2.4.2 Rigid-body transformations . . . 16
- 2.5 Robot model . . . 18
- 2.6 Improved robot model . . . 22
- 2.7 Implementation and performance evaluation . . . 23

3 Collision detection 25
- 3.1 Separating Axis Test (SAT) implementation . . . 25
  - 3.1.1 Motivation . . . 25
  - 3.1.2 SAT basics . . . 25
  - 3.1.3 Optimizations of the inequalities . . . 27
  - 3.1.4 Caching . . . 27
  - 3.1.5 Performance evaluation . . . 28
- 3.2 Spatial (hierarchical) decomposition . . . 29
  - 3.2.1 Outline of the data structure . . . 30
  - 3.2.2 Discretization of the configuration space . . . 30
  - 3.2.3 Sample labeling . . . 34
  - 3.2.4 Linear quadtrees . . . 35
  - 3.2.5 Linear 5D quadtree . . . 42
  - 3.2.6 Linear 4D quadtree . . . 46
  - 3.2.7 Guarantees . . . 54
  - 3.2.8 Symmetries . . . 55
  - 3.2.9 Performance evaluation . . . 57

4 Conclusions and future work **69**

Bibliography **71**

Appendix **73**

A Project execution **73**
A.1 Implementations in the NJ-controller 73

List of Figures

1.1 A typical application. . . . 1
1.2 A typical application to pack cookies into boxes. . . . 2
1.3 Scara robot . . . 3
1.4 Delta3-R robot . . . 4
1.5 Current application . . . 5
1.6 Desired application . . . 5

2.1 Delta3-R robot . . . 12
2.2 Universal joint used in Delta3-R robot . . . 13
2.3 Rotational joint used in Delta3-R robot . . . 13
2.4 Definition of the Machine-Coordinate-System of a robot. The x-axis is aligned with the conveyor belt and the origin O in placed on the center point of the top frame of the robot. . . . 14
2.5 Different bounding volumes. . . . 15
2.6 An example of the robot model, drawn as OBBs around the links of the robot. Further, the MCS is defined on the center of the top plane. . . . 17
2.7 An example of three attached bodies. . . . 18
2.8 A schematic version of a Delta3-R robot. The links are drawn as lines instead of boxes for clarification. . . . 19
2.9 Parameters to describe an oriented bounding box. . . . 20
2.10 A schematic version of one arm of a Delta3-R robot projected on the xz-plane. . . . 20
2.11 The different body frames used to describe the pose of c_2 . . . 21
2.12 Comparision of the performance of the robot model and the improved version. The timings are measured for constructing two robot models. . . . 24

3.1 Two OBBs A and B that are not in collision from the perspective of separating axis L. . . . 26
3.2 Comparison of the performance of SAT and SAT-cache on two test cases. . . . 29
3.3 Comparison of the performance of SAT and SAT-cache on two test cases together with the robot model. . . . 29
3.4 The connections between the various components in the data structure. . . . 31
3.5 The configuration space: a cylinder and a frustum cone . . . 31
3.6 Determining the circle radius in the frustum cone . . . 33
3.7 Grid placed over a circle from $\mathcal{CS}$ in which red points are omitted in the sample set. 33
3.8 Comparision of the performance of labeling samples with a different amount of threads. . . . 35
3.9 A quadtree and the corresponding subdivision of a square. . . . 36
3.10 A quadtree and the corresponding subdivision indexed by location codes. . . . 37
3.11 A quadtree and the corresponding subdivision indexed by location codes. . . . 39
3.12 Interval that is monotonic (left) and an interval that is not monotonic (right). . . . 43
3.13 An example of how a 2D slice looks like. The black squares represent collision configurations and the white squares represent non collision configurations. . . . 46

3.14 Two 2D slices $\mathcal{S}_1$ and $\mathcal{S}_2$ that are matched to each other (3.14a). The add-differences are depicted on the left in 3.14b and the sub-differences are depicted on the right. 48
3.15 A 2D slices $\mathcal{S}$ and the corresponding bit array $\mathcal{B}$). 49
3.16 A quilt with yy'-slices of the configuration space. Black boxes represent collision configurations and white boxes represent no-collision configurations. 51
3.17 A quilt with zz'-slices of the configuration space. Black boxes represent collision configurations and white boxes represent no-collision configurations. 52
3.18 A configuration c that is being treated as point q. The dashed dotted enclosed area is a collision free area. 55
3.19 A xy-projection of two Delta3-R robots. 56
3.20 A xy-projection of two Delta3-R robots, rotated 180° around c_1. 57
3.21 The amount of memory (10^4 bytes) of the linear 4D quadtree when using yy'- or zz'-slices. 60
3.22 The amount slices (groups) of the linear 4D quadtree when using yy'- or zz'-slices. 60
3.23 The amount of memory (10^6 bytes) of the linear quadtree (6D), linear 5D quadtree (5D), linear 4D quadtree (4D) when using yy'-slices and $k = 64$. 61
3.24 Comparision of the performance of SAT (including robot model in red) and the linear quadtree variants. 62
3.25 Three slices $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$. 64
3.26 Binary classification for the linear quadtree with different settings for the robot model. 65

A.1 An overview of the project implementation inside the robotics system. 74

List of Tables

3.1 The three test cases with begin and end position for the TCP of R_1 and R_2. The positions are defined in the MCS of the corresponding robot. 57
3.2 Parameters for the memory and time experiments. 58
3.3 Results for the linear quadtree (6D), linear 5D quadtree (5D), linear 4D quadtree (4D) with yy'-slices. Time refers to the construction time of the data structure. The memory requirements are computed as stated in Observation 3.2.1, 3.2.2 and 3.2.3. 59
3.4 Results for the linear quadtree (6D), linear 5D quadtree (5D), linear 4D quadtree (4D) with zz'-slices. Time refers to the construction time of the data structure. The memory requirements are computed as stated in Observation 3.2.1, 3.2.2 and 3.2.3. 59
3.5 Results for the linear quadtree (6D), linear 5D quadtree (5D), linear 4D quadtree (4D), when $k = 64$. Time refers to the construction time of the data structure. The memory requirements are computed as stated in Observation 3.2.1, 3.2.2 and 3.2.3. 60
3.6 Average query time in microseconds for $k = 32$ (584,416 collision configurations). The trajectories of all test cases consist of 6000 positions and thus 6000 queries are performed. 61
3.7 Average query time in microseconds for $k = 64$ (22,583,625 collision configurations). The trajectories of all test cases consist of 6000 positions and thus 6000 queries are performed. 61
3.8 Parameters for the binary classification experiments. 63
3.9 Accuracy information for testcase T_1 with $D_a = 30$ and $H_a = 82$ and $k = 32$. . . . 65
3.10 Accuracy information for testcase T_1, with $D_a = 17.5$ and $H_a = 70$ and $k = 32$. . . 65
3.11 Accuracy information for testcase T_2 with $D_a = 30$ and $H_a = 82$ and $k = 32$. . . . 66
3.12 Accuracy information for testcase T_2, with $D_a = 17.5$ and $H_a = 70$ and $k = 32$. . . 66
3.13 Accuracy information for testcase T_3 with $D_a = 30$ and $H_a = 82$ and $k = 32$. . . . 66
3.14 Accuracy information for testcase T_3, with $D_a = 17.5$ and $H_a = 70$ and $k = 32$. . . 67

Listings

3.1 Linear quadtree definition 41
3.2 Linear 5D quadtree definition 44
3.3 Linear 4D quadtree definition 50

Chapter 1

Introduction

In the packaging industry there are many tasks that are inherently repetitive and suitable for automation. One of the common applications is packing objects into some container. For example, a factory that produces cookies wants to pack them into small boxes that one can buy at the supermarket. These boxes are then packed into larger boxes and later palletized such that they are suitable for transport. Usually, the objects arrive on a conveyor belt and must be placed in containers that are positioned on another nearby conveyor belt. The task of placing the objects, here cookies or small boxes, into their larger containers is of a repetitive nature. Robots are perfectly suited for these kinds of tasks and it is therefore not surprising that the packaging industry uses a lot of robots in these applications. An example of an application is depicted in Figure 1.1.

Figure 1.1: A typical application.

The packaging industry is always looking for a system that increases the amount of objects that can be packed during a fixed time frame. One way to do this is to place more robots along the conveyor belts. One of the problems with this approach is that placing more robots along a conveyor belt usually means that the robots are placed more closely to each other (the length of the

conveyor belt can usually not be extended too much). This gives rise to the problem that robots could collide with each other, for example when they move towards each other. To overcome this problem, one would like to create a system that makes sure that the robots do not collide during operation. A primitive in this system is to be able to detect when robots are actually colliding. This problem is called *collision detection* and will be the topic of this thesis. The system in which this primitive will be used is build on top of a real-time operating system, which is common in modern controllers for robotics applications.

1.1 Packaging robots

The applications in the packaging industry vary widely, i.e. packing (soft) food is different from packing metal screws. Further, it is not even fixed what components are used, even for a specific application. This is due to the (physical) limitations of the factory in which the system must operate. Most applications, however, share some properties. In Figure 1.2 we have depicted a schematic version of a typical application. Here, C_1 and C_2 denote the two conveyor belts and R_1, R_2 and R_3 denote the mounting positions of the three robots. Now, the shared properties between the applications are:

- Objects arrive on a conveyor belt, C_1 that is always moving, i.e. there are always some objects arriving.
- Close to the conveyor belt(s), one or multiple robots (R_1, R_2 and R_3) are mounted statically to the environment.
- The objects have to be packed into some container (for example a box), usually located on a second conveyor belt C_2. This conveyor belt can either be always moving, or stopped while packing the objects into the container.
- One tries to keep the size of the system, that consists of the conveyor belts, robots and other parts as small as possible.

In such packaging applications, there are two types of robots that are most often used: *Scara* and (parallel) *Delta* robots, see Figure 1.3 and Figure 1.4, respectively. The Delta robot has several clear advantages over the Scara robots. It is more easy to build them, even by some of the customers themselves. Further, they are considered the fastest robots in their field. This because they do not include their motors on the moving parts, but instead on a statically mounted part. Another reason is that a movement of the robot is described by a combination of several (parallel) arms. More about the mechanics is written in Section 2.2. Because they are fast and relatively

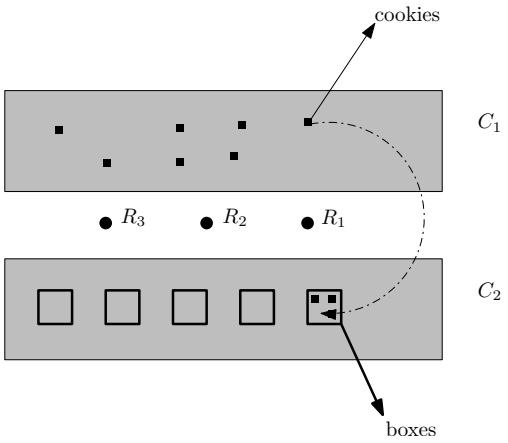


Figure 1.2: A typical application to pack cookies into boxes.

Figure 1.3: Scara robot

cheap to build, they are very popular and widely used in the packaging industry. Both the Scara and Delta robots share some properties in the context of being used in the packaging industry:

- The robots are mounted statically to the environment and can thus not move or rotate.
- The robots are placed in such a way that possible collisions can only occur between a few robots (typically two or three).
- The robots can be represented by a few simple volumes (as explained in detail in Section 2.5).

Although the theory in this thesis has been developed such that it is applicable to multiple (different) robot types, we will focus throughout on the Delta3-R type. This because they are considered to be the fastest robots in their field, and thus from this point of view the hardest to perform collision detection on. Further, the model representation of Delta3-R robots and other robots (like the Scara robots) will not differ too much: they both consist of a few rectangular pieces, as is discussed in Section 2.4.1.

The performance of a system, consisting of multiple conveyor belts and robots, is often considered as the throughput of the system. Here, the throughput is the amount of objects that are packed into their containers during a given time period. The higher the throughput, the more objects that are packed. In the packaging industry, there is always a demand for systems that increase throughput. Common approaches to increase the throughput are using more and faster robots, multiple conveyor belts and performing the packing process in multiple phases.

Using more robots for a conveyor belt would assume that the conveyor belt must be made larger, which is usually not possible. The amount of space that can be occupied by a system consisting of conveyor belts and robots is fixed due to other constraints in the factory. Therefore, one wants to place the robots in such a way that a minimal amount of space is wasted.

In Figure 1.5, there are two conveyor belts, C_1 and C_2, depicted. Next to them, there are two robots, R_1 and R_2. The disks centered at the position of R_1 and R_2 denote the area that is reachable by the corresponding robot. The *uncovered area* is the area on the conveyor belts that

Figure 1.4: Delta3-R robot

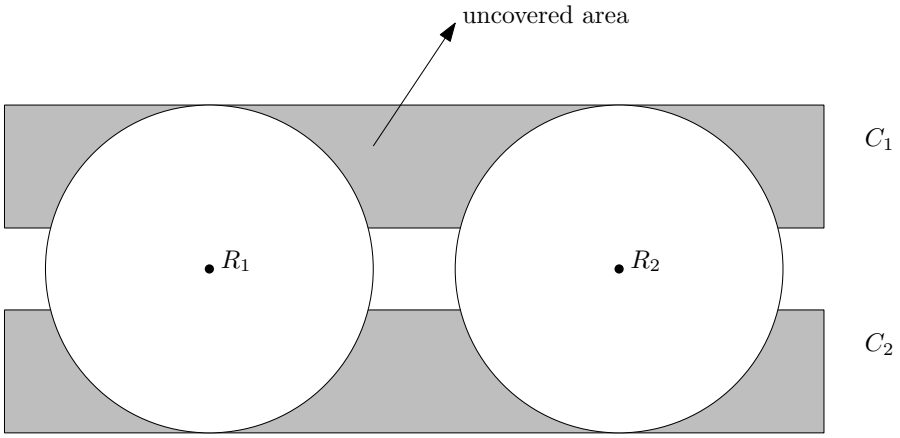


Figure 1.5: Current application

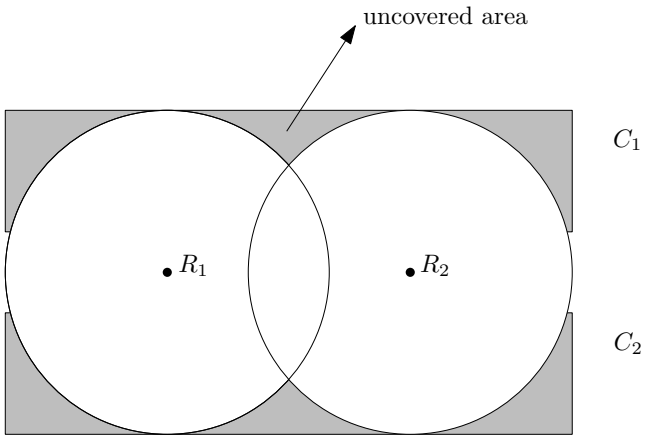


Figure 1.6: Desired application

is not reachable by any of the robots. This area is considered as a waste of space, and since the space in these systems is scarce one tries to minimize this area.

One approach to minimize this area is to place the robots closer to each other, where the reachable areas might even overlap. This approach is depicted in Figure 1.6. In this approach the uncovered area is smaller and the conveyor belts themselves can also be made smaller (or one could place more robots at the same conveyor belts). This approach, however, has the drawback that the robots can possibly collide with each other. This requires efficient collision detection techniques and this will be the topic studied in this thesis.

1.2 Collision detection

The problem of collision detection can be viewed in its most basic form as the development of a function c: $o_1 \times o_2 \rightarrow \{\text{TRUE}, \text{FALSE}\}$, which given two objects o_1 and o_2 reports whether there is a collision (represented by TRUE) or not (represented by FALSE). This problem arises not only in the field of robotics, but for example also in computer graphics, physical simulations and haptic feedback.

There are many variants of this problem. One variant is to perform collision detection for a set of objects. One can think of an application of this problem as a collision detection primitive in a computer game. There can be thousands of objects moving in a shared space and one wants to compute which objects will collide (in the next frame or eventually). If the movements of

the objects can be described by some motion function (and this is known in advance), one can consider the use of *kinematic data structures* (KDSs). These data structures will prune collision detection for pairs that cannot collide (up to the next frame, for example), by looking at their motion functions.

An important aspect for collision detection algorithms is the complexity of the objects, which can vary greatly. Some objects are very simple and can be represented by a few *bounding volumes* (BVs), while others require a large number of them. A bounding volume is a volume that describes the object that it contains by some approximation, but for which collision detection (or another problem) is easier to solve. See Section 2.5 for a more detailed description of bounding volumes. An approach to overcome the problem of having to perform collision detection for a large amount of bounding volumes, is to use a *bounding volume hierarchy* (BVH). In this hierarchy, an object is represented by multiple levels of BVs, that represent the object in an increasing amount of detail. So at the first levels one uses large BVs, and at the lowest level small BVs are used. These small BVs then represent the (parts of the) object they contain more accurately. To solve the problem of collision detection, one starts with the big BVs and tests them for collisions. The BVs for which collision is detected are inspected further and the other BVs are skipped. This results in a search in which the BVs become increasingly smaller and in which the search will be focused on the area in which the collision actually happens.

Another variant is to perform collision detection in multiple phases, a *broad phase* and *narrow phase*. In a broad phase one looks at whether objects will collide by using BVs that are coarsely approximating the objects. If the objects do not collide, more precise collision detection does not have to be performed. If the objects collide, one applies a more precise technique to find out whether the objects really collide.

Recently, a newer variant that has attracted a lot of attention is collision detection for deformable objects. An example of deformable objects are clothes that change their shape under the action of external forces. The problem of *self collision detection* also arises here. In this problem one wants to find out whether an object collides with itself, for example if some clothes are dropping on the floor.

Collision detection can be performed under various 'circumstances'. One can perform collision detection for dynamic objects or for static objects, i.e. in which objects are allowed to move or not. Further, one can perform collision detection in a real-time setting or not. This real-time setting arises in applications in computer graphics, haptic feedback and robotics.

In addition to determining whether there is a collision or not, it is sometimes also useful to know the distance between objects or the amount of penetration of two objects (i.e. how 'far' two objects intersect). For example in haptic technology one wants to know the amount of pressure that is applied to some surface. Knowing the amount of pressure, the surface can perform a reaction that is of the same proportion. Knowing the distance between two objects is also useful in applications where one wants to prune (expensive) collision detection. If the distance is large, one can defer an expensive method for (precise) collision detection.

For a more thorough overview of 'general' collision detection literature, we refer to [10, 14, 16, 18].

Most novel techniques focus on problems where, for example, there are a huge amount of objects that can possibly collide with each other. Another focus is on complex objects, in which one applies a BVH or more sophisticated algorithms. In many techniques one also assumes that the objects can freely move in some space. Despite the extensive research in these fields, existing methods are not ideal for our setting because we have relatively simple objects in a rather static environment. On the other hand, as we will see next, the real-time nature of the application asks for extremely fast methods.

1.3 Real-time controller

A solution for the collision detection problem has to be implemented in a robotics software system. We will describe this system shortly, since it poses additional constraints on possible approaches. The robotics software system runs on an embedded system, which is called a *NJ-controller*. The NJ-controller contains a modern processor and limited memory.

The robotics software system is implemented on top of a real-time operating system and operates in certain *ticks* (or phases). A tick is a certain amount of time in which a status update for the entire system is performed. For example, when the system is connected to multiple robots, the robotics software system computes an updated target position for every robot during every tick. The duration of such a tick is typically 1 millisecond, but can be configured to be 1/2, 2 or 4 milliseconds. Throughout the thesis we will assume a duration of 1 millisecond, as this is the most commonly used value. During a tick, the system first performs some preprocessing after which the actual computations can take place. After all computations have been performed, the system will do some postprocessing. All these phases have to complete within the duration of a tick, and thus within 1 millisecond.

A natural question in performing collision detection is to wonder how the trajectories of the robots look like. If the trajectories have a structure in some sense, one could try to exploit this. This kind of information is unfortunately not available in the robotics software system. The only information that is available is the previous position and the current position. There is thus no information available about how a robot is moving along a trajectory, nor what consecutive transformations will be applied. This is the case because the robotics software system has to be applicable to a lot of different (practical) scenarios. Instead, the system offers a few primitives from which an application engineer can compose a suitable trajectory. Therefore, the collision detection primitive should also abstract from the trajectories and focus on 'static' situations. The most suitable way to do this is to perform collision detection every tick, since an application engineer can then use this information in computing the next location in the trajectory.

During every tick, as stated above, there is limited time to do computations. In addition to this, the available time will also be limited by other tasks that have to be executed. Therefore, a timing constraint of 1 millisecond for the collision detection primitive is too unrealistic, since this will leave no room for other (essential) tasks. A timing constraint for the collision detection primitive that is realistic is that it should be able to perform its computations within a comparable time frame of other, currently used, primitives. These primitives, such as solving kinematics problems and generating parts of trajectories, currently perform their computations within ~40 microseconds.

Due to this strict timing constraint, some primitive operations cannot be performed on the robotics software system. For example, it is not possible to read from or to write to a file on a disk. This implies that for every data structure that we will design it must fit in primary memory.

Next to the strict timing requirement, the NJ-controller contains a limited amount of memory. Therefore, there is also the (strict) requirement that every data structure that we will design, and that must fit in primary memory, is not larger than tens or a few hundreds of megabytes.

1.4 Problem statement

As stated before, the applications in the packaging industry, together with the commonly used robots share some properties. These properties describe a particular class of applications and robots. In this thesis we will focus the problem of collision detection on this particular class of applications and robots. In the rest of the thesis, we will focus all descriptions on the Delta3-R robot. For each topic, we will then describe how the theory can be expanded to the other types of robots that are used in this class.

The problem treated in this thesis can now be formalised as follows: *'Develop a procedure that solves the collision detection problem for a pair of (Delta3-R) robots. The procedure must be implemented into the NJ-controller. The NJ-controller puts the additional constraints that the procedure must be able to perform its computations close to 40 microseconds and within a memory limit of tens (or a few hundreds) of megabytes'.*

1.5 Related work

As already stated in Section 1.2, there are many variants of the collision detection problem. In this section, we will give an overview of relevant literature: the literature that focuses on real-time dynamic collision detection. The real-time aspect of this problem pops up in fields as computer graphics, physical simulations and haptic technology. The first two fields, require generally a real-time constraint of ~30 milliseconds. Although this constraint is too unrealistic for our context, much of the work is relevant. The third field, haptic feedback, generally has the real-time constraint of ~1 milliseconds, which means that it is much closer to our real-time constraint.

One inherent property of real-time collision detection is that it could possibly exploit *temporal coherence*. Temporal coherence means in this context that an object does not move 'too' much during two consecutive collision detection queries. When collision detection is performed every millisecond, the corresponding objects have almost the same position and orientation. This can be exploited for better performance. Most algorithms described in this section, natively support this or can be extended to support some form of temporal coherence.

There are several well-known algorithms that solve the collision detection problem in the context of real-time constraints. One of the most popular ones is the GJK algorithm, which is an abbreviation for Gilbert–Johnson–Keerthi algorithm [5]. The algorithm computes simplices out of the vertices of the Minkowski difference of two objects. If such a simplex then contains the origin, it can report that the two objects are in collision and also report the penetration distance. If the simplex does not contain the origin, the algorithm can conclude for that case that there is no collision. The advantages of this approach are that it is fast and the code base is small. Some practical problems occurring while implementing the algorithm, such as numerical stability are solved and mentioned by a later paper [17]. Also some performance improvements are included in that version.

Another approach was taken in [13]. Here, one uses an incremental approach (instead of a simplex-based method) to identify the two closest features (faces, edges and vertices) on the two objects. When they are found, one can compute the distance between the two objects. Now, during a consecutive collision detection query, one expects that the closest features will be the same, or nearby features. This makes the algorithm very fast when there is temporal coherence, as it only needs to update the closest feature pair to nearby features.

The disadvantage of the approach in [13] is that it does not support the case of penetrating polyhedra (and it suffers from numerically robustness issues). There are some approaches that can be taken to circumvent this problem, as is discussed in [15]. In that paper, another approach is described, the Voronoi-clip method. It uses Voronoi-regions to determine what the two closest features can be, along with other useful observations for a speedup. The method provides support for penetrating objects and is robust. The performance is comparable to that of the GJK algorithm.

A third approach, which we discuss in more detail in Section 3.1, is the SAT algorithm, which is an abbreviation for Separating Axis Test [6]. In the literature it is also referred to as the Separating Axis Theorem. This algorithm solves the collision detection problem by identifying a separating (hyper)plane between two convex objects.

Some work [11] has been done to store information efficiently about the environment of robots, so that for example path planning can be performed. One of the applications is that one captures the space where robots can move and parts of the space where movement is not possible, because of obstacles. The work in [8] is one of the examples for this and gives a nice overview of relevant

literature. Although that we use a similar technique to speedup collision detection, the setting in which this is done is different. Current literature tries to model and store the physical world in which the robots are used, while we store the *configuration space* of two robots (as explained in Section 3.2.2). Further, the focus of the literature is largely on obstacles for possibly moving robots, while we have statically mounted robots in which other robots are the obstacles.

1.6 Results and organization

In this thesis we consider the problem of dynamic real-time collision detection for packaging robots. In Chapter 2 we first describe how to model the robot such that this abstraction is suitable to perform computations on. Next, in Section 3.1, we first present a simple existing but efficient and exact technique to perform collision detection. We also show that most of the computational cost for this method is actually to compute the robot model. This motivates another approach: one that makes use of a precomputed data structure, which is discussed in Section 3.2. It turns out that this technique is superior regarding computational cost, but at the cost of increased memory requirements. Since memory is also scarce in the real-time system, multiple variants of this precomputed data structure are presented. These variants try to split the 'high' dimensionality of the data structure in order to save memory. We present the results of experiments in which it is shown that the memory requirements are indeed much lower, while not increasing the computational cost too much.

Chapter 2

Robot mechanics and model

In this chapter we will give an overview and detailed description of the important concepts regarding the robots. First, in Section 2.1, we introduce some terminology. Next, in Section 2.2, we present details about the physical construction of the robot, which imposes limitations on the movements it can make. Further, to describe the position of (parts of) the robot we need to make use of some fixed coordinate systems, which are defined in Section 2.3. Next, since we need to abstract the exact shape of the robot, we investigate what methods can be used to do this in Section 2.4. Further, computing the actual robot model is described in Section 2.5. To improve efficiency, we developed an improved version of the robot model, as explained in Section 2.6. Finally, we present the performance of the two robot models in Section 2.7.

2.1 Terminology

A robot exists in physical three-dimensional space, which is denoted by $\mathcal{W} = \mathbb{R}^3$. The *pose* of the robot, or a certain part of it, is a description of the position and orientation in $\mathcal{W}$. Formally, one can define a pose as $p = [x, y, z, \alpha, \beta, \gamma]$, where (x, y, z) represents the position in $\mathcal{W}$, α represents the rotation around the z-axis, β represents the rotation around the y-axis and γ represents the rotation around the x-axis.

The *configuration* of a robot is a collection of parameters that are needed to describe the poses of all parts of the robot. The (minimal) amount of parameters that are needed is also called the *degrees of freedom* (DOFs) of a robot. If one could, for example, describe a robot by knowing the values of three angles, the robot would have three DOFs, and thus a three-dimensional configuration.

2.2 Robot mechanics

The Delta3-R robot (depicted in Figure 2.1) consists of two parallel frames: the *top frame* (depicted by (1)) and the *bottom frame* (depicted by (2)). The top frame is fixed statically to the environment. This implies that the top frame will never translate or rotate. Further, the bottom frame will not rotate in any direction and can thus only translate in the physical space.

The two frames are connected by three *arms*. Each arm consists of two links: the *upper link* (3) and the *lower link* (4). These two links are connected to each other and the corresponding frames by three *joints*. Joints are mechanical parts that allow connected objects to rotate or translate relative to each other, according to the precise type of joints that is used. The joints are depicted by (5), (6) and (7). In this robot design two types of joints are used: *universal joints* and *rotational joints* (depicted in Figure 2.2 and Figure 2.3, respectively). The rotational joints are used in (5), while the universal joints are used in (6) and (7). The rotational joints are *active* joints, which means that the angle between the two connected parts is actively influenced by the motor that is attached at this joint. The universal joints are *passive* joints, which means that the angle

Figure 2.1: Delta3-R robot

between the two connected parts is not directly influenced by a motor. More precisely, the angle is influenced indirectly by motors connected to other joints. When two links are connected by the rotational joint, they can describe a circle, i.e. the joint introduces one degree of freedom. The universal joint allows the two links to describe a sphere, hence the joint introduces two degrees of freedom.

At the bottom of the robot (2), a *tool tip* is attached. We will refer to the tool tip as *TCP* (Tool Center Point) throughout this thesis. For each specific application one can attach an appropriate *tool* to this TCP. The tool can then be designed to have a suitable geometry to pick up the objects on the conveyor belt. Rotation of the tool is possible if the robot is equipped with an additional fourth axis, the *rotational axis* (8).

The robot further contains three motors (9). They are connected via a rotational joint (5) with the upper link (3). The motor can only make a circular movement, which will cause the upper link to rotate and to get a new pose. The rotation of this upper link will also cause the lower link of the same arm to have a different pose. In fact, it will affect the poses of the lower links of the other two arms as well. Because of this, the exact pose of the lower link of a single arm cannot be determined by only the amount of rotation one motor has taken. The exact pose of all lower links can only be determined by the amount of rotation all three motors have taken (and not by fewer motors, since the last motor will affect the pose of all lower links). In other words, to describe the exact pose of the entire robot, one needs at least three parameters: the three rotations of the joints. When the additional rotational axis is attached, an extra parameter is needed to describe the amount of rotation around this axis, resulting in a total of four parameters. This implies that the robot has three DOFs and thus three-dimensional configurations. With the additional rotational axis this will be four DOFs and four-dimensional configurations.

The TCP of the robot can be moved by using a combination of movements of the three motors; by applying rotations on (some of) the three motors. It can be assumed that the motors will rotate exactly as commanded, and that the TCP of the robot will therefore also move as commanded. More specifically, if the TCP is commanded to move to position (x, y, z), then we assume that the TCP will be exactly at (x, y, z) after the motors executed their commands. This is also called *open loop control*. In practice this will not be the case, as the motors will always make a slight error in their movement. This error is, however, in practical scenarios small enough to be ignored.

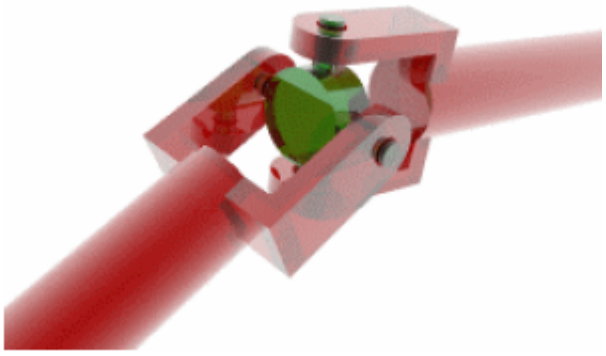


Figure 2.2: Universal joint used in Delta3-R robot

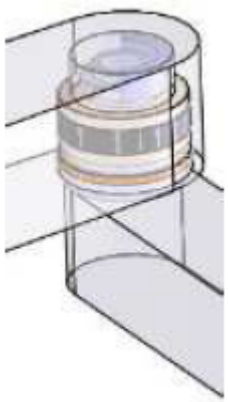


Figure 2.3: Rotational joint used in Delta3-R robot

2.3 Coordinate systems

To specify certain positions, poses of parts of a robot and trajectories, one needs a reference coordinate system. The origin of this coordinate system must be defined on a fixed position with

a fixed orientation. For every robot, a *Machine-Coordinate-System* (MCS) is defined, in which these concepts can be expressed. The origin of a MCS is the center point of the top frame as depicted as point O in Figure 2.4. The orientation is defined as follows. Suppose we would define a coordinate system in two dimensions. The x-axis of the MCS is aligned with the arrow a on the conveyor belt (see Figure 2.4). This arrow represents the direction of the movement of the conveyor belt. Furthermore, the y-axis is orthogonal with the x-axis. Since the coordinate system must be defined for a three-dimensional space, we define the additional z-axis to be orthogonal to both the x-axis and y-axis. In other words, the x- and y-axis are defined in the plane that is described by the conveyor belt.

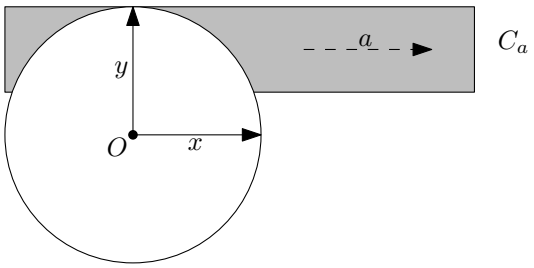


Figure 2.4: Definition of the Machine-Coordinate-System of a robot. The x-axis is aligned with the conveyor belt and the origin O in placed on the center point of the top frame of the robot.

With only different MCSs for every robot, it is impossible to define the location of one robot with respect to another robot. To solve this problem, we define a *World-Coordinate-System* (WCS). This coordinate system is equal to one of the MCSs. Thus, the origin of this coordinate system is placed on top of the origin of one of the MCSs and the orientation is also aligned. Defining the WCS like this has the benefit that the information about one robot does not have to be transformed to this WCS (since the corresponding transformation matrix will be the identity matrix). To describe the position and orientation of another robot, we apply a predefined transformation that encodes the necessary translation and rotation.

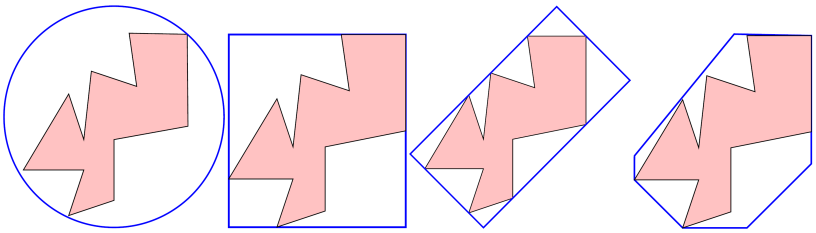
Let p_{MCS} be a point declared in the coordinate system MCS, which is a different coordinate system than the WCS. Now to describe its position in the WCS, we apply the following transformation matrix ${}^{WCS}T_{MCS}$. In this transformation matrix, α denotes a rotation around the z-axis and x_t, y_t and z_t denote the translation from the origin of MCS to the origin of the WCS.

$$ {}^{WCS}T_{MCS} = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & x_t \\ \sin\alpha & \cos\alpha & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

Now, to describe p_{MCS} in the WCS, we need to apply the following transformation.

$$ p_{WCS} = {}^{WCS}T_{MCS} p_{MCS} $$

A constraint on the placement of the robots (relatively to each other), and therefore also on the positions and orientations of the different MCSs is the following. The robots can never be rotated around the x- or y-axis (relative to each other). Since there are no restrictions on the positions of the robots, the transformation between an arbitrary MCS and the WCS can always be described by the transformation matrix ${}^{WCS}T_{MCS}$, by using the appropriate values for α, x_t, y_t and z_t.



(a) Sphere (b) AABB (c) OBB (d) k-DOP

Figure 2.5: Different bounding volumes.

2.4 Geometric modeling and transformations

In this section we describe what modeling techniques we use for the robot, and how to compute the actual robot model. This process is two-fold: first we must define a certain robot model. Next, we must compute the model with respect to some fixed coordinate system.

2.4.1 Geometric modeling

The actual robot has a fairly complex shape. It consists of multiple arms, which consists of multiple links (that also consists of multiple parts). Further it contains motors, springs, frames and joints. The mechanics of the robot and the electrical components used to control the robot are imprecise so that exact poses cannot be guaranteed. The error that is made is small, so usually this is ignored and does not cause any problems in practice. The robot model can exploit this by representing a simplified version of the exact, physical robot. It can approximate the structure to enable faster computations on the model. This approximation will then 'forget' about uninteresting parts and model the interesting parts.

There are various options to model the robot. One important decision to make is whether it is desirable to have a solid or boundary representation. In 2D, an example of a solid representation is a disk, while the boundary representation is a circle. Since we are here interested in merely whether two robots collide, we are interesting in their boundary representations. If their boundary is intersecting, the solid variant would also be intersecting. Technically, one could argue that the opposite is not true, if one object is entirely inside another object. Modelling them as solid objects will still produce a collision, while having only a boundary representation will not produce a collision. In this thesis we make the assumption that robots cannot be inside of each other (which is physically also true) and therefore focus on boundary representations.

A virtual model of a complex shape or structure can be constructed by splitting this shape or structure into several smaller parts. For example in the case of the Delta3-R robots, the virtual model can be build by modeling the arms, frames and tools separately. In fact, the interesting parts of the robot to model are the lower links and the tool part. This because the upper links cannot collide due to constraints on the possible placements. To model these parts, one can use bounding volumes (BVs). A bounding volume is a closed volume that contains a certain part (or certain parts) of a object. In this case, a bounding volume can be seen as a volume that contains part(s) of the robot. The benefit of using a bounding volume to represent a certain part of the robot is that this bounding volume can be described more easily and would, ideally, provide a *tight fit*. A tight fit means that the bounding volume closely represents the object it contains, and does not contain too much space that is not part of the object. We thus want to find boundary volumes for the lower links and tool part of the robot.

In Figure 2.5 we depicted several popular bounding volumes. When looking at the options from left to right (so from (a) to (d)), one can see that the bounding volumes fit the shape more tightly. On the other hand, intersection tests become more expensive (computationally), so one has to make a tradeoff between these two contradictory properties.

Spheres. If one wants to compute intersection between two spheres, one can look whether the distance between the center points is less than the sum of the two radii. If this is the case, the two spheres are in collision, otherwise they are not in collision. This check is simple and can be implemented very efficiently. The drawback of using spheres as bounding volume is that they contain a lot of space that might not belong to the object, i.e. no tight fit.
Axis-Aligned Bounding Boxes. An axis-aligned bounding box (AABB) is a box of which the axes are aligned with a coordinate system. The axes of the box are thus not necessarily aligned with the orientation of the object it contains and can therefore contain a lot of space that is not part of the object. The intersection tests are cheap, however, since one only has to look at the distance between the center points of two boxes, and the sizes of the boxes in the corresponding dimensions. In the case of packaging robots, AABBs might contain too much space that is not part of the robot. This will result in detection of collisions in cases where there is no collision.
Oriented Bounding Boxes. An oriented bounding box (OBB) is the same kind of bounding volume as an AABB, but with its axes aligned to the orientation of the object it contains. This provides a very tight fit for shapes that are more or less rectangular. Using OBBs for modelling packaging robots will provide a very tight fit, since their parts are rectangular.
k-DOPs. A k-DOP, or a *discrete oriented polytope* is a generalization of the AABB. A k-DOP is a convex polytope (in 3D, a convex polyhedron) that is defined by k planes by which a space is bounded. In case of an AABB, the space is bounded by six axis-aligned planes (parallel to the faces of the box). By using more than six planes, one can obtain a better approximation of the object this bounding volume contains. This will, however, also increase the cost of computing whether there is a collision between two k-DOPs. In case of packaging robots, the k-DOPs do not provide a better fit than OBBs, since the best fit will be a box that is oriented in the same way as a part of the robot.

A different way of modeling the robot is to use an hierarchy of bounding volumes. One can think of such a hierarchy as a (binary) search tree in which the nodes represent bounding volumes. Every child of a node then represents a smaller part of the robot (within the current bounding volume) and models this part with a smaller, but tighter bounding volume. Such a hierarchy usually uses more bounding volumes, but provides a better fit. The overhead while searching is not significant, since the search structure can prune the uninteresting parts easily. In the case of only using OBBs for the links, there is no need to further subdivide these bounding volumes. The OBBs already provide a tight fit of the links.

Looking at the pros and cons of the different bounding volumes, it is clear that OBBs provide the best fit. Therefore, we will construct the robot model with OBBs. See Figure 2.6 for an example of the robot model of both upper and lower links drawn together with the top and bottom frames. The actual used robot model for the Delta3-R robot will only contain OBBs for the lower links, as the upper links cannot collide. The tool part of the robot will not be modeled. This because for current applications this is not needed—the tool part is small enough to not collide with the entire robot. For future applications and bigger tools, one can always add the appropriate bounding volumes to the robot model.

Also to model other packaging robots, such as Scara robots, OBBs are perfectly suited. These robots consist also of a few rectangular pieces, which are modeled nicely when using OBBs.

2.4.2 Rigid-body transformations

Before we explain how exactly we model the robot, we must introduce some terminology to be able to express transformations on that model. For a more precise and formal introduction, we refer to [2] and [12]. We borrow notation from [12] to describe the necessary notions.

Recall that we defined the physical world as $\mathcal{W} = \mathbb{R}^3$. In this world, we can describe (parts of) our robot $\mathcal{A}$. To do that, we must first introduce *frames*. A frame is a fixed coordinate system,

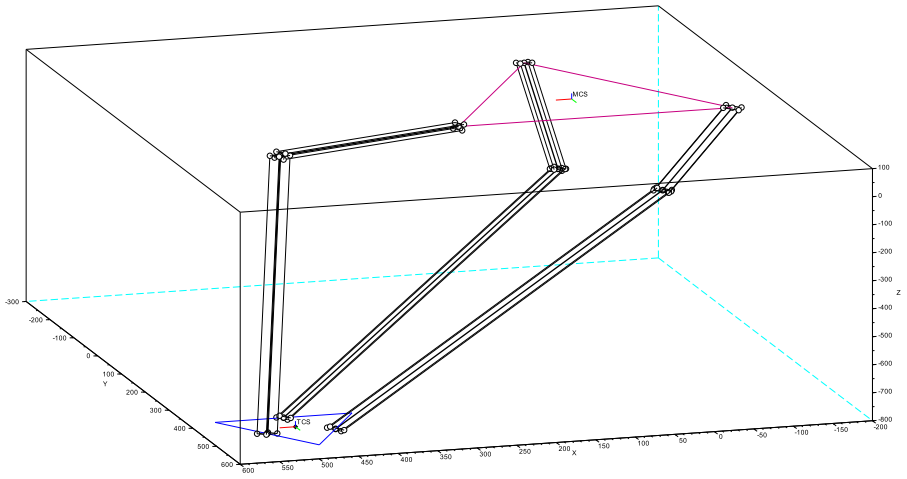


Figure 2.6: An example of the robot model, drawn as OBBs around the links of the robot. Further, the MCS is defined on the center of the top plane.

such as the MCS and WCS which are defined in Section 2.3. The *world frame*, describing $\mathcal{W}$ is usually described by the origin and base vectors of $\mathbb{R}^3$, but in our case we will use the origin and base vectors of the WCS. Next to the world frame, we can also define *body frames*: 'local' frames that are attached to some part of the robot, a *body* of the robot. Some situations can be described more easily in such a local frame. One problem that arises with the use of body frames is that it does not describe objects in the world frame anymore, which is usually required. To solve this problem, one uses transformations to describe a body frame into the world frame.

Let $\mathcal{A}(q)$ denote that robot $\mathcal{A}$ is placed at configuration q in some frame. This means that the *reference point* of $\mathcal{A}$ is placed at the position denoted by q and that $\mathcal{A}$ is aligned accordingly. Now, assume that we place $\mathcal{A}$ at q in a body frame. We want to express $\mathcal{A}$ in the world frame. To do this, we must transform the configuration q to be a configuration q' defined in the world frame. This can be done by using the homogeneous transformation matrix M, as depicted below. Here x, y, z, α, β and γ represent the transformation from the body frame to the world frame. It describes a transformation of a rotation α around the x-axis, a rotation β around the y-axis and a rotation γ around the z-axis. After these rotations, it describes a position (x, y, z) in this new orientation. By applying this matrix to a pose in the body frame, we get the corresponding pose in the world frame. This process can be repeated for entire chains of bodies, provided that the corresponding transformations are known.

$$M(x,y,z,\alpha,\beta,\gamma) = \begin{pmatrix} \cos\gamma\cos\beta & \cos\gamma\sin\beta\sin\alpha - \sin\gamma\cos\alpha & \cos\gamma\sin\beta\cos\alpha + \sin\gamma\sin\alpha & x \\ \sin\gamma\cos\beta & \sin\gamma\sin\beta\sin\alpha + \cos\gamma\cos\alpha & \sin\gamma\sin\beta\cos\alpha - \cos\gamma\sin\alpha & y \\ -\sin\beta & \cos\beta\sin\alpha & \cos\beta\cos\alpha & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In Figure 2.7 we have depicted an (2D-)example in which there are three bodies, B_1, B_2 and B_3. The bodies B_1 and B_2 are attached at joint j_2 and bodies B_2 and B_3 are attached at joint j_3. Let j_1 be the reference point of the robot. Let the angle that body B_2 makes with B_1 be $\alpha_{1,2}$ and the angle between body B_3 and B_2 be $\alpha_{2,3}$. Further, let j_1 be at position (x_1, y_1) in the world frame (the frame that is depicted in the figure). The transformation matrix that describes j_1 in the world frame is now given by the 2D homogeneous transformation matrix ${}^{W}_{B_1}M(x_1, y_1, 0)$.

Further, the x-component of the distance between j_1 and j_2 is x_2 and the y-component is y_2. So, its corresponding transformation matrix is ${}^{B_1}_{B_2}M(x_2, y_2, \alpha_{1,2})$. Consistently, the x- and y-components of the distance between j_2 and j_3 are x_3 and y_3. The corresponding transformation matrix is ${}^{B_2}_{B_3}M(x_3, y_3, \alpha_{2,3})$. Now, to express the pose of j_3 in the world frame, we apply the following chain of transformations:

$$ {}^{W}_{B_1}M(x_1, y_1, 0){}^{B_1}_{B_2}M(x_2, y_2, \alpha_{1,2}){}^{B_2}_{B_3}M(x_3, y_3, \alpha_{2,3}) $$

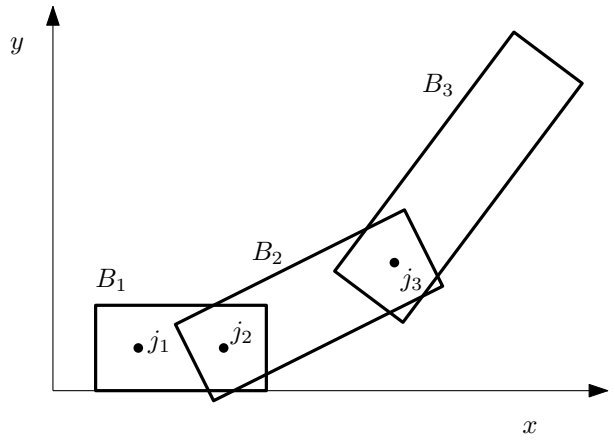


Figure 2.7: An example of three attached bodies.

This technique can be applied for any number of connected bodies and it is the technique we will use to construct the robot model in Section 2.5. Apart from this technique, they are other ways to express poses at the end of a kinematic chain in a world frame, such as Euler angles or the Denavit-Hartenberg notation. For this, we refer to [2].

2.5 Robot model

In order to compute the representations of the OBBs for the robot, we need to apply several transformations that simulate the pose of the parts of the robot. The first transformation is already explained in Section 2.3, which is the transformation from the WCS to the MCS. We will apply the other transformations as explained in Section 2.4.2.

In Figure 2.8 a Delta3-R robot is depicted, without the fourth rotational axis. Recall that the robot model consist of three OBBs that are placed around the line segments G_iE_i, for $i \in \{1, 2, 3\}$. The F_i points represent the positions of the three motors in the real robot. The angle between the dashed line l and F_2G_2 is defined as θ_2. This angle θ_2 represents the amount of rotation the motor placed at F_2 has taken (angles θ_1 and θ_3 are defined analogously). This angle, together with angles θ_1 and θ_3, will determine the exact pose of the robot.

The points G_i represent the joints between the upper link (described by F_iG_i) and lower link (described by G_iE_i). They are, as explained in Section 2.2, passive and have two degrees of freedom. The β_i angles represent a rotation around the local y-axis, and the γ_i angles represent a rotation around the local z-axis. The degrees of freedom of the joints at G_i are depicted by the two angles β_2 and γ_3. The points E_i represent the joints that connect the lower links to the bottom frame.

A convenient representation for an OBB is the following. We represent an OBB (depicted in Figure 2.9) by defining its center point p, its local coordinate system consisting of the vectors A_x, A_y and A_z and the half-dimensions D_a, H_a and W_a. To compute the representation of the OBB at the lower links, we need the centerpoint and the base vectors of the local coordinate system.

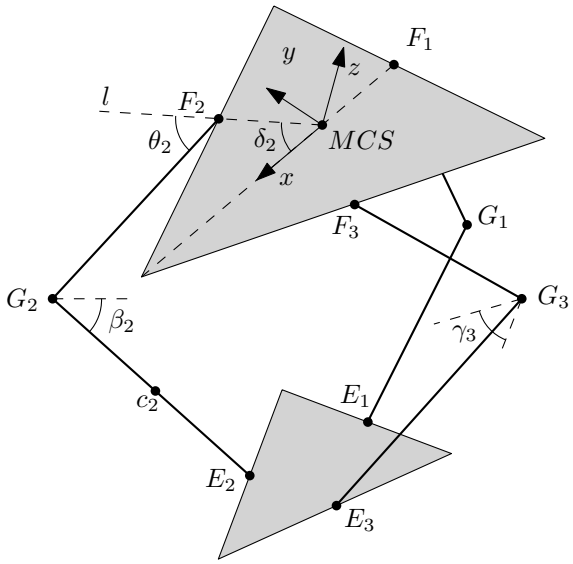


Figure 2.8: A schematic version of a Delta3-R robot. The links are drawn as lines instead of boxes for clarification.

We will describe this procedure for only one arm, as the rest is symmetric under a rotation around the z-axis. For example, in Figure 2.8, we need for the second arm of the robot point c_2 as the centerpoint and the base vectors are defined as follows. The x-axis must be aligned with the lower link, so it must be aligned with G_2E_2. The y- and z-axis are somewhat more difficult to define. For the z-axis we first look at the situation in Figure 2.10. In this figure, one arm of the Delta3-R robot is projected on the xz-plane. As already defined, the x-axis is aligned with the G_2E_2 line segment. Now, the z-axis is defined as being orthogonal to the x-axis in this xz-projection. The direction is obtained by rotating the initial z-axis of the WCS positively twice around the y-axis, as is visualized the difference in the body frame axes in by Figures 2.11b, 2.11c and 2.11d. The result of this is depicted in Figure 2.10.

The y-axis is rotated, from the definition of the MCS, only once, as depicted in the Figures 2.11d and 2.11e. This is apart from the rotation for the second and third leg, which is depicted in Figures 2.11a and 2.11b.

To compute the representation of the OBB for the lower link, we make use of body frames, as explained in Section 2.4.2. For clarity, we will use a body frame for every step in describing the pose of c_2. The body frames are generated as follows:

1. Transform from WCS to MCS, see Figure 2.11a.
2. Transform the origin of the body frame to F_i position and align the x-axis with the direction of F_iG_i, see Figure 2.11b.
3. Transform to include the rotation of the motor, around the y-axis (depicted as θ_i), see Figure 2.11c.
4. Transform the origin of the body frame to G_i position, including the rotation around the y-axis (depicted as β_i), see Figure 2.11d.
5. Transform to include the rotation around the z-axis (depicted as γ_i), see Figure 2.11e.
6. Transform the origin of the body frame to the center point (depicted as c_i) of the lower link (orientation stays the same), see Figure 2.11f.

The corresponding transformation matrices are the following. Here we use the notation defined in Section 2.4.2.

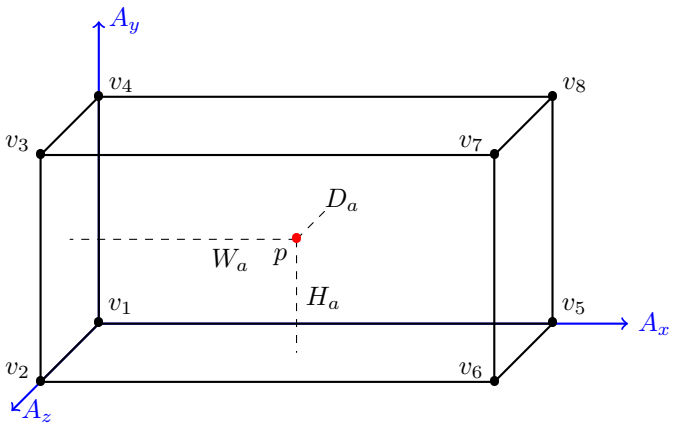


Figure 2.9: Parameters to describe an oriented bounding box.

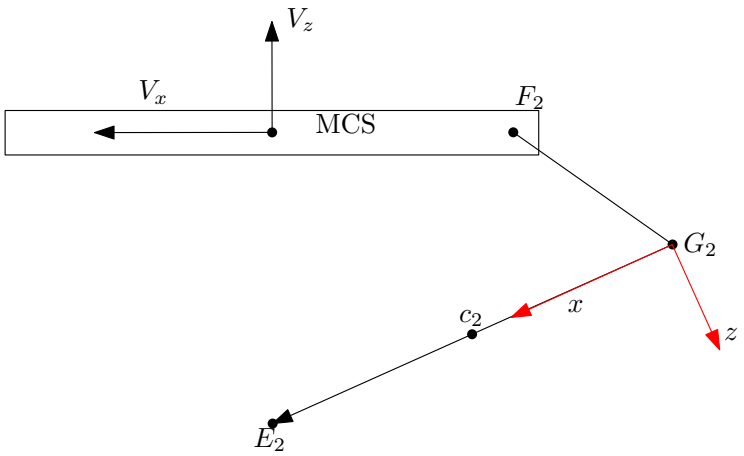


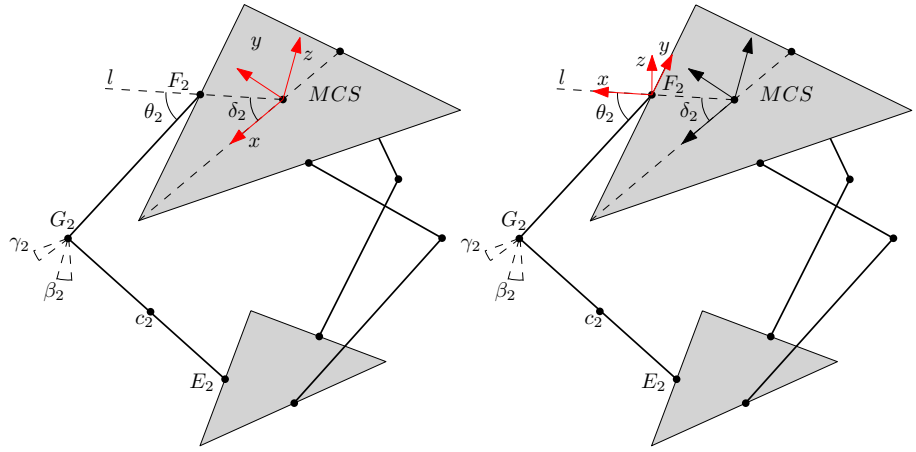
Figure 2.10: A schematic version of one arm of a Delta3-R robot projected on the xz-plane.

1. $M_1 = M(x, y, z, 0, 0, \gamma)$.
2. $M_2 = M(F_i(x), F_i(y), F_i(z), 0, 0, \delta_i)$, where δ_i is the amount of rotation around the z-axis.
3. $M_3 = M(0, 0, 0, 0, \theta_i, 0)$, where θ_i is the amount of rotation the motor takes (depicted as θ_2).
4. $M_4 = M(|F_iG_i|, 0, 0, 0, \pi - \beta_i - \theta_i, 0)$, where β_i is the amount of rotation between the upper and lower link (depicted as β_2) and $|F_iG_i|$ represents the length of the upper link.
5. $M_5 = M(0, 0, 0, 0, 0, \gamma_i)$, where γ_i is the amount of rotation of the lower link around the local z-axis (depicted as γ_3).
6. $M_6 = M(|G_iE_i|/2, 0, 0, 0, 0, 0)$, where $|G_iE_i|$ represents the length of the lower link.

Now, the pose of point c_i can be computed as follows.

$$M_1 M_2 M_3 M_4 M_5 M_6$$

From this pose we know the position and orientation. The center point of the OBB will be the location of point c_i and the local coordinate system will correspond to the orientation from this pose. The (half)width (parameter W_a) will correspond to half of the length of $|G_iE_i|$ and the (half)height H_a and (half)depth D_a of the OBB are configurable.



(a) Pose of the first body frame.

(b) Pose of the second body frame.

(c) Pose of the third body frame.

(d) Pose of the fourth body frame.

(e) Pose of the fifth body frame.

(f) Pose of the sixth body frame.

Figure 2.11: The different body frames used to describe the pose of c_2

2.6 Improved robot model

In order to perform collision detection for a pair of robots, we have to construct a pair of robot models. In Chapter 3, there are two different approaches described regarding when exactly to compute these robots models. Nonetheless, in order to perform collision detection, the robot models have to be computed at some point. It is therefore important that the models are computed efficiently. Looking at the approach used in Section 2.5, there is some room for improvement. One can, for example, speed up computing the product of the entire chain of matrices (M_1 till M_6, in Section 2.5) by simplifying the resulting equations.

These simplified equations represent the transformation from the world frame (at the MCS), to the body frame placed at c_i. From these equations we can compute directly the position of c_i. Also the base vectors of the body frame at that position can be obtained. The base vectors are then obtained by computing the unit vectors in this body frame. For the x-component we will subtract point $(1, 0, 0)$ in this body frame from the origin (center point c_i), and do the same for the y- and z-component by using the offset points $(0, 1, 0)$ and $(0, 0, 1)$.

The simplified equations are parameterized by a vector $q = (q_x, q_y, q_z)$ that denotes whether we want to have the origin (center point c_i), or the base vectors. To compute the robot model, we first compute the center point of the OBB, with $q = (0, 0, 0)$. Then we compute the three base vectors for the OBB by subtracting the points given by $q = (1, 0, 0)$, $q = (0, 1, 0)$ and $q = (0, 0, 1)$ from the center point. Further, it takes as input the β_i and γ_i angles (see Section 2.5), where $i \in \{1, 2, 3\}$ for legs one, two and three, respectively. Next, $G_{i,x}$ denotes the x-component of point G_i, for leg i (and a similar symbol for the y- and z-components). Finally, the equations take as input parameter l, that describes the distance from point G_i to c_i, which is half the distance between G_i and E_i. The simplified equations are stated below. The equations are obtained with the help of the computer algebra system Maxima 5.31.2.

Leg 1:

$$p_x = \cos(\gamma_1)q_y + \sin(\gamma_1)q_x + \frac{\sin(\gamma_1)l}{2} + G_{1,x} \tag{2.1}$$

$$p_y = -\sin(\beta_1)q_z - \cos(\beta_1)\sin(\gamma_1)q_y + \cos(\beta_1)\cos(\gamma_1)q_x + \frac{\cos(\beta_1)\cos(\gamma_1)l}{2} + G_{1,y} \tag{2.2}$$

$$p_z = -\cos(\beta_1)q_z + \sin(\beta_1)\sin(\gamma_1)q_y - \sin(\beta_1)\cos(\gamma_1)q_x - \frac{\sin(\beta_1)\cos(\gamma_1)l}{2} + G_{1,z} \tag{2.3}$$

Leg 2:

$$\begin{aligned} p_x = {} & \frac{\sqrt{3}\sin(\beta_2)q_z}{2} + q_y\left(\frac{\sqrt{3}\cos(\beta_2)\sin(\gamma_2)}{2} - \frac{\cos(\gamma_2)}{2}\right) \\ & + q_x\left(\frac{-\sin(\gamma_2)}{2} - \frac{\sqrt{3}\cos(\beta_2)\cos(\gamma_2)}{2}\right) - \frac{\sin(\gamma_2)l}{4} - \frac{\sqrt{3}\cos(\beta_2)\cos(\gamma_2)l}{4} + G_{2,x} \end{aligned} \tag{2.4}$$

$$\begin{aligned} p_y = {} & \frac{\sin(\beta_2)q_z}{2} + q_y\left(\frac{\cos(\beta_2)\sin(\gamma_2)}{2} + \frac{\sqrt{3}\cos(\gamma_2)}{2}\right) \\ & + q_x\left(\frac{\sqrt{3}\sin(\gamma_2)}{2} - \frac{\cos(\beta_2)\cos(\gamma_2)}{2}\right) + \frac{\sqrt{3}\sin(\gamma_2)l}{4} - \frac{\cos(\beta_2)\cos(\gamma_2)l}{4} + G_{2,y} \end{aligned} \tag{2.5}$$

$$p_z = -\cos(\beta_2)q_z + \sin(\beta_2)\sin(\gamma_2)q_y - \sin(\beta_2)\cos(\gamma_2)q_x - \frac{\sin(\beta_2)\cos(\gamma_2)l}{2} + G_{2,z} \tag{2.6}$$

Leg 3:

$$p_x = \frac{\sqrt{3}\sin(\beta_3)q_z}{2} + q_y\left(\frac{-\sqrt{3}\cos(\beta_3)\sin(\gamma_3)}{2} - \frac{\cos(\gamma_3)}{2}\right) + q_x\left(\frac{\sqrt{3}\cos(\beta_3)\cos(\gamma_3)}{2} - \frac{\sin(\gamma_3)}{2}\right) - \frac{\sin(\gamma_3)l}{4} + \frac{\sqrt{3}\cos(\beta_3)\cos(\gamma_3)l}{4} + G_{3,x} \quad (2.7)$$

$$p_y = \frac{\sin(\beta_3)q_z}{2} + q_y\left(\frac{\cos(\beta_3)\sin(\gamma_3)}{2} - \frac{\sqrt{3}\cos(\gamma_3)}{2}\right) + q_x\left(\frac{-\sqrt{3}\sin(\gamma_3)}{2} - \frac{\cos(\beta_3)\cos(\gamma_3)}{2}\right) + \frac{\sqrt{3}\sin(\gamma_3)l}{4} - \frac{\cos(\beta_3)\cos(\gamma_3)l}{4} + G_{3,y} \quad (2.8)$$

$$p_z = -\cos(\beta_3)q_z + \sin(\beta_3)\sin(\gamma_3)q_y - \sin(\beta_3)\cos(\gamma_3)q_x - \frac{\sin(\beta_3)\cos(\gamma_3)l}{2} + G_{3,z} \quad (2.9)$$

2.7 Implementation and performance evaluation

The robot models are implemented as described in Section 2.5 and 2.6. Also described there, is that the robot models need the angles θ_i, β_i and γ_i and the points G_i and E_i. This information is retrieved from (parts of) the direct and inverse kinematics engine. This engine implements solutions for the *direct kinematics* and *inverse kinematics* problem. The direct kinematics problem is: given the three θ_i angles, compute the position of the TCP of the robot. The inverse kinematics problem is the opposite: given the TCP position of the robot, compute the three θ_i angles. We will not describe the details of (parts of) the implementation of the kinematics engine here. Some extra parameters are needed to execute the solutions for the direct and inverse kinematics problems (these will be the same throughout the entire thesis, since we use one particular robot):

1. The distance between the MCS origin and the F_i points: $f = 100$.
2. The distance between F_i and G_i: $r_f = 150$.
3. The distance between G_i and E_i: $r_e = 400$.
4. The distance between E_i and the TCP: $e = 40$.

The next experiment is performed on the NJ-controller. The experiment is performed *outside* the robotics software system, to avoid some complications while testing. The performance measurements are reliable, however, since the experiment is run on exactly the same hardware as is used for the robotics software system. The setup of the experiment is as follows: compute the (improved) robot model 100,000 times on the same TCP position. Then the time we report for each robot model is the average time it takes to compute the robot model. In Figure 2.12, one can see that there is an significant improvement between the 'normal' robot model and the improved version.

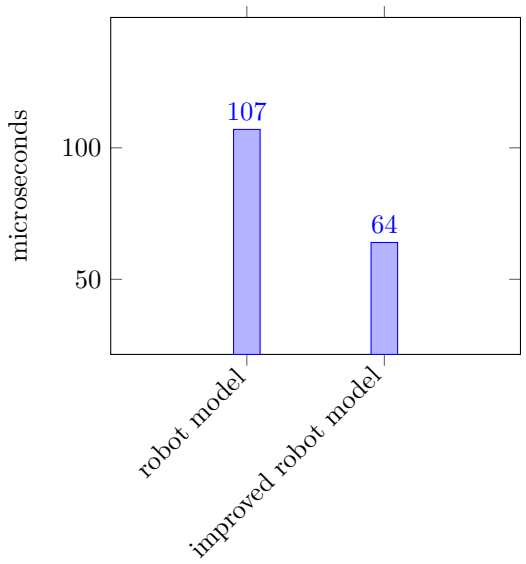


Figure 2.12: Comparision of the performance of the robot model and the improved version. The timings are measured for constructing two robot models.

Chapter 3

Collision detection

In this chapter we will describe the various algorithmic approaches that we developed to solve the collision detection problem. The approaches use the robot model, as described in Section 2.5. In Section 3.1 we will describe the Separating Axis Test, a simple but efficient algorithm that solves the collision detection problem. Further, in Section 3.2 we will describe a novel approach, based on spatial hierarchical decomposition. This method will make use of a precomputed data structure, so that it performs much better than the approach using the Separating Axis Test.

3.1 Separating Axis Test (SAT) implementation

The robot model consists of three OBBs, one for each lower link. To compute whether two robot models collide, we have to check for each pair of OBBs (one OBB from each robot model) whether they intersect. We use the *Separating Axis Test* (SAT) in order to compute whether two OBBs, who are arbitrarily translated and rotated, intersect. This technique is also commonly referred to as the *Separating Axis Theorem*. It is particularly suited for the collision detection problem, since it can be efficiently applied to the case of OBBs. In fact, Gottschalk et al. [6] claims that it is possible to compute within 200 arithmetic operations whether an intersection takes place. Further, the technique is numerically robust and does not suffer from geometric degeneracies. A drawback of using this technique is that it is only applicable to polygonal models (i.e. not to cylinders, cones, spheres, etc.). The robot model, however, does not use such polygonal models. We will base the following sections on the work of Gottschalk et al. [6].

3.1.1 Motivation

There are several algorithms suited to perform collision detection for two robots models that consist of OBBs. For example, the GJK algorithm [17] would be suited for this. Further, closest features techniques and even linear programming approaches can solve this problem. The reason why we choose the SAT algorithm to perform collision detection is because of its efficiency (it is faster then previous mentioned approaches [6]), numerical stability and robustness regarding geometric degeneracies. Also, certainly not the least factor, it is simple to implement.

3.1.2 SAT basics

The technique is based on the following observation. Let A and B be two OBBs. A *separating plane* is a plane that separates the space in such a way that A will be on one side of the plane and B on the other side. If A and B do not intersect, it is possible to define such a separating plane between them and if they do intersect, it is not possible to define such a plane. Using this observation, we can reduce the problem of computing whether there is a collision to the computation of such a separating plane.

Luckily, the computation of such a plane for the case of two OBBs is not very expensive. A separating plane can be generated as the normal of a *separating axis*. Line L is a separating axis, if the projections of A and B on L are disjoint. Thus, if we can find one separating axis (where the projections of A and B on are disjoint), we also found a separating plane. For a pair of OBBs there are 15 possible separating axes. Six of the axes are defined by the base vectors $V_A = \{A_x, A_y, A_z\}$ and $V_B = \{B_x, B_y, B_z\}$ of the local coordinate systems of A and B. This because the separating planes of interest are parallel to the faces of the boxes. These separating planes are defined by the base vectors of the boxes. The remaining nine axes are defined by the cross product of pairs of these base vectors (so by elements of $V_A \times V_B$). Each pair of these base vectors spans a plane, that allows to generate separating planes that are able to detect non-collision in cases where the first six separating planes (generated by the base vectors) would detect collision.

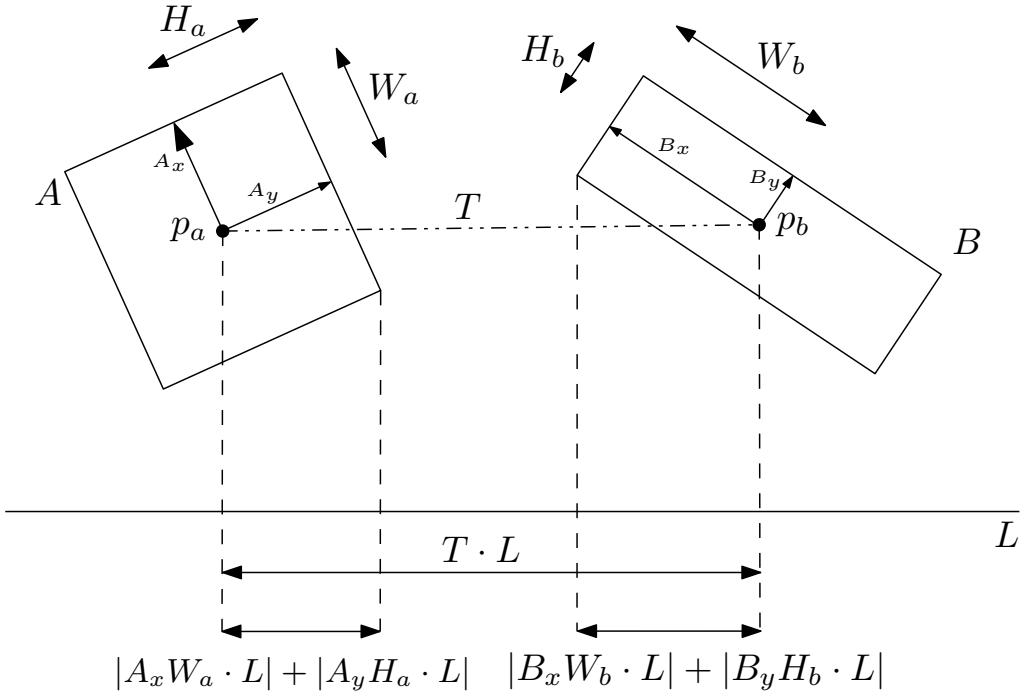


Figure 3.1: Two OBBs A and B that are not in collision from the perspective of separating axis L.

Before we describe the SAT algorithm in detail, we will first introduce some notation that will be used in this section and in Section 3.1.3. With Uv, where U is a vector and v is a scalar, we denote the multiplication between them. Next, $U \cdot V$, where $U = (u_1, u_2, ..., u_n)$ and $V = (v_1, v_2, ..., v_n)$ are both vectors, denotes the dot product: $\sum_{i=1}^{n} u_i v_i$. Finally, $U \times V$, where both $U = (u_1\boldsymbol{i} + u_2\boldsymbol{j} + u_3\boldsymbol{k})$ and $V = (v_1\boldsymbol{i} + v_2\boldsymbol{j} + v_3\boldsymbol{k})$ are vectors, denotes the cross product: $(u_2v_3 - u_3v_2)\boldsymbol{i} + (u_3v_1 - u_1v_3)\boldsymbol{j} + (u_1v_2 - u_2v_1)\boldsymbol{k}$.

We will describe the SAT algorithm for a 2D case; the algorithm for the 3D case follows the same structure. A 2D scenario is depicted in Figure 3.1. Let T denote the distance between the two center points p_A and p_B and let L denote one of the possible separating axes as described above. We project half of A and B on L, denoted by $|A_x W_a \cdot L| + |A_y H_a \cdot L|$ and $|B_x W_b \cdot L| + |B_y H_b \cdot L|$, respectively. Further, we project T on L. Now, if half of the projection of A added to half of the projection of B on L results in a longer interval than the projection of T on L, A and B are in collision from the perspective of L. If this is is not the case, A and B will not be in collision. In this latter case, a plane orthogonal to L will represent a separating plane. Note, that if we find 'evidence' for a collision from the perspective of L, it does not have to be the case that A and B actually collide—there might be another separating axis. Only if all possible separating axes

result in 'evidence' of a collision we can conclude that A and B actually collide. The checks are represented by the following inequality.

$$|T \cdot L| > |A_x W_a \cdot L| + |A_y H_a \cdot L| + |B_x W_b \cdot L| + |B_y H_b \cdot L|$$

To extend this to the 3D case, we need to include the third dimension in the projections of A and B, resulting in the following inequality.

$$|T \cdot L| > |A_x W_a \cdot L| + |A_y H_a \cdot L| + |A_z D_a \cdot L| + |B_x W_b \cdot L| + |B_y H_b \cdot L| + |B_z D_b \cdot L|$$

3.1.3 Optimizations of the inequalities

The inequalities described above can be modified to require less calculations. For these optimizations we need the following observations, which are reported in [9]. Also, in [9], the proofs of the observations and the resulting inequalities (using all observations) are given.

The inequalities include several dot products, and in some cases the two vectors that are the operands of these dot products are orthogonal. For example, when assuming L to be A_x, we obtain the following inequality.

$$|T \cdot A_x| > |A_x W_a \cdot A_x| + |A_y H_a \cdot A_x| + |A_z D_a \cdot A_x| + |B_x W_b \cdot A_x| + |B_y H_b \cdot A_x| + |B_z D_b \cdot A_x|$$

Here, we have that A_x is orthogonal to both A_y and A_z, so that these dot products result in zero. Further, the dot product A_x and itself will be one. This results in the following inequality (other inequalities can be optimized in the same way).

$$|T \cdot A_x| > W_a + |B_x W_b \cdot A_x| + |B_y H_b \cdot A_x| + |B_z D_b \cdot A_x|$$

Observation 3.1.1. $sA \cdot (B \times C) = sC \cdot (A \times B)$

Using Observation 3.1.1 and the fact that the cross product between a vector and itself is always zero, we can rewrite the inequality

$$\begin{aligned}|T \cdot (A_x \times B_x)| > |W_a A_x \cdot (A_x \times B_x)| + |H_a A_y \cdot (A_x \times B_x)| + |D_a A_z \cdot (A_x \times B_x)| + \\ |W_b B_x \cdot (A_x \times B_x)| + |H_b B_y \cdot (A_x \times B_x)| + |D_b B_z \cdot (A_x \times B_x)|\end{aligned}$$

to the following inequality.

$$|T \cdot (A_x \times B_x)| > |H_a A_z \cdot B_x| + |D_a A_y \cdot B_x| + |H_b A_x \cdot B_z| + |D_b A_x \cdot B_y|$$

Observation 3.1.2. $T \cdot (A_x \times B_x) = (T \cdot A_z)(A_y \cdot B_x) - (T \cdot A_y)(A_z \cdot B_x)$

Observation 3.1.2 is defined in the context of the specific case in which $L = A_x \times B_x$, but it can be applied also to the other eight cases in which L is given by the cross product of two base vectors. Using this observation, the left hand sides of the inequalities can be rewritten so that they can be calculated more efficiently.

3.1.4 Caching

Another possible optimization for the SAT algorithm exists when one takes a look at the context in which this algorithm will be used. As already explained in Section 1.5, we can exploit temporal coherence. In this particular case, we can exploit temporal coherence by caching the *separating witness*. The separating witness is (in this case) the separating axis L for which we detected that there is no collision (hence, the existence of the *separating* axis/witness). We use this axis L in

the next query before testing the other fourteen axes. Now, assuming that the robot models did not transform too much, it is likely that this axis L is still a separating axis, and we do not have to try the fourteen other axes. This will not improve the worst case performance of the algorithm, but rather the average case performance.

Since the robot model consist of three OBBs, we have to perform nine tests: one for every pair of OBBs from the two robots. In our implementation of SAT, we stop the algorithm and return that there is a collision when one of these nine pairs result in a collision. We thus do not test the other (possibly up to eight) pairs when there is already a collision detected. The pair that is responsible for a collision can also be cached: in the next query one can first test this pair. If the pair is still in collision, the other eight pairs do not have to be tested. Otherwise, one can just check the other eight pairs. This makes that our cache performs in two layers. The first layer caches a possible already colliding pair. If there is no colliding pair, we just test all pairs. The second layer then stores which axis is a possible separating axis. The first layer thus tries to speed up situations in which there is a collision, while the second layer tries to speed up situations in which there is no collision.

3.1.5 Performance evaluation

We measured the performance of the SAT algorithm in two variants on two different test cases. The two variants are the SAT algorithm without or with the caching of the separating witness (see Section 3.1.4); we refer to them as SAT and SAT-cache, respectively. The first test case, the 'practical' test case, is a test case in which two robots are moving towards each other without colliding. We call this test case the practical test case, because in a realistic setting the robots will not collide. If one would simulate the robots to move in such a way that they will collide at some point, it makes sense that the robots are stopped immediately when they collide (so the test case would contain only one collision). The second test case, the 'collision' test case, is a case in which the robots are moving towards each other and colliding with each other. The last part of the trajectory of the robots results in collision, so that many invocations of the SAT algorithm will result in the detection of a collision. The results are depicted in Figure 3.2.

One can make two main observations from this graph. The first is that in a case in which there are collisions, using a cache really can help. The execution time for the SAT algorithm drops from five microseconds to two microseconds, because the 'right' arm that is in collision can be found immediately in the first layer of the cache. This saves testing fifteen equations for the other arms each. The second observation is that in the practical test case, in which the robots do not collide, maintaining the cache results in worse performance. This is because there are no collisions, and thus the algorithm cannot benefit from the first layer in the cache. It needs, however, to check and maintain this cache. The second layer in the cache, that tries to skip some axis to test cannot overcome the maintenance of the two layers of caching.

In Figure 3.3, we depicted the same graph as in Figure 3.2, but included the time needed to compute the two robot models. What immediately becomes clear in this graph is that the SAT algorithm performs very fast, but that there is a large overhead from the robot models.

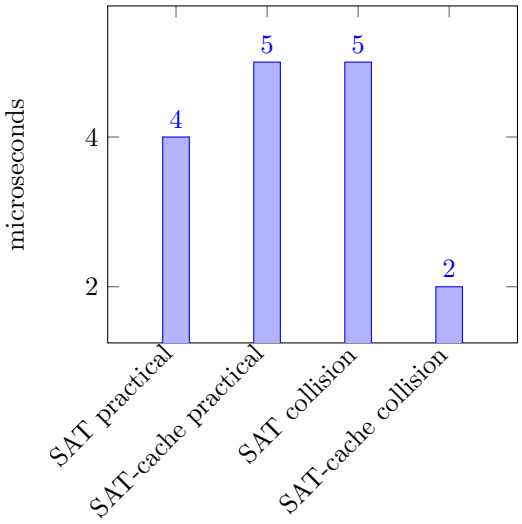


Figure 3.2: Comparison of the performance of SAT and SAT-cache on two test cases.

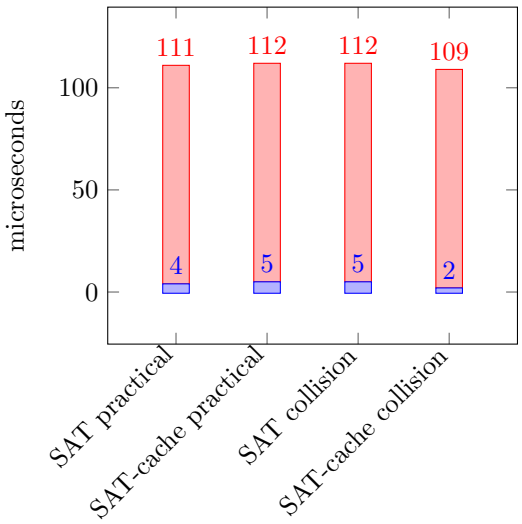


Figure 3.3: Comparison of the performance of SAT and SAT-cache on two test cases together with the robot model.

3.2 Spatial (hierarchical) decomposition

In some cases, one can speed up performance of algorithms by making use of a precomputed data structure. For example, when one is given an array of integers, one can speed up searching for a particular element by first sorting this array. After investing time to sort the array, a binary search can be performed to find the element. The problem of finding a particular element is now divided in two phases: a precomputation phase and a query phase. Since usually the precomputation phase has a higher computational complexity and the query phase has a lower computational complexity, this approach pays off in a situation where one has a *multi query problem*. In a multi query problem one expects to have many queries, so that the higher computational complexity for constructing the data structure can be recovered by having multiple queries, each of a lower computational complexity.

This idea of using a precomputed data structure can also be applied for the collision detection

problem. Here we also have this multi query problem. Specifically, during every tick (see Section 1.3) a new query for the collision detection problem has to be answered, while the setup of the robots and their environment stays largely the same.

The drawback of the method described in Section 3.1 is that it requires to construct the robot model (for two robots) for every query. As explained in Section 3.1.5, constructing the robot model is expensive in comparison with the SAT algorithm.

To avoid constructing the robot model for every query we want to store this information in some data structure. This data structure can then be precomputed on a more powerful computer, so that during system operation we only have to query it. To exploit this idea even more, we could also already invoke the SAT algorithm on the robot models and store the answer to the query in the data structure. Now, during system operation, one has to query the data structure and the result is directly the answer to the collision detection problem. Another advantage of this approach is that it exploits the multi query problem we have here: we can re-use this data structure for every query during system operation.

3.2.1 Outline of the data structure

Before describing every part of the data structure in detail, we first describe how everything will fit globally together. The approach can be split in two different parts, as already mentioned above: the precomputation- and query-phase. The query-phase consists basically of one part: a search procedure to retrieve information from the data structure. This will be explained in Section 3.2.4. The precomputation-phase consists of several parts, as depicted in Figure 3.4. These parts are:

- Discretization of the configuration space ('Sampler'), explained in Section 3.2.2.
- Classification of the produced samples ('Label samples'), explained in Section 3.2.3.
- Generation of indices for the produced samples and their storage in the data structure ('Data structure construction'), explained in Section 3.2.4.
- Compression of the data structure ('Data structure construction'), explained in Section 3.2.4.

3.2.2 Discretization of the configuration space

We will first look at what the data structure actually must contain. Ideally, one has to ask the data structure during system operation if the robots at their current position are in collision. Recall (as explained in Section 2.1) that one can describe the pose of a robot by defining the position of its TCP. It is thus convenient if the data structure is indexable by the TCP positions of both robots. This implies that we need to store for 'all' possible TCP positions for two robots whether they result in a collision (or not).

To describe the data that we want to store in more detail, we must first formalize some notions. Recall that a configuration describes the pose of two robots. In this context, we will denote with a configuration $c = (x, y, z, x', y', z')$ the positions of the TCPs of two robots, placed at (x, y, z) and (x', y', z'), respectively. All possible configurations together describe the *configuration space*, $\mathcal{CS} \subset \mathbb{R}^6$, of the two robots. The data structure should store for which configurations there will be a collision and for which not.

A first problem that arises is how to capture 'all' possible configurations for two robots. The configuration space is a continuous space, which we cannot store in a limited amount of memory. Therefore, we have to discretize this $\mathcal{CS}$ into something suitable for storage. We perform such a discretization by taking samples of $\mathcal{CS}$. We apply deterministic sampling by sampling according to a grid, i.e. every sample has a predefined distance to its neighbours. We choose this approach to be able to provide guarantees about the size of the error the discretization will make, as explained in Section 3.2.7. Further, this approach has the advantage of having samples from 'every' part of

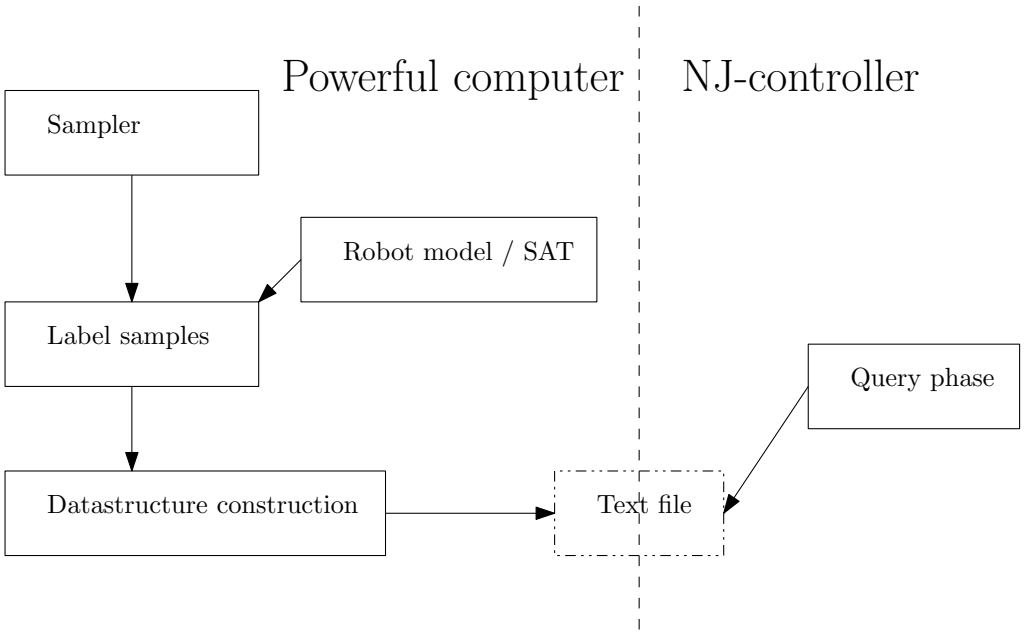


Figure 3.4: The connections between the various components in the data structure.

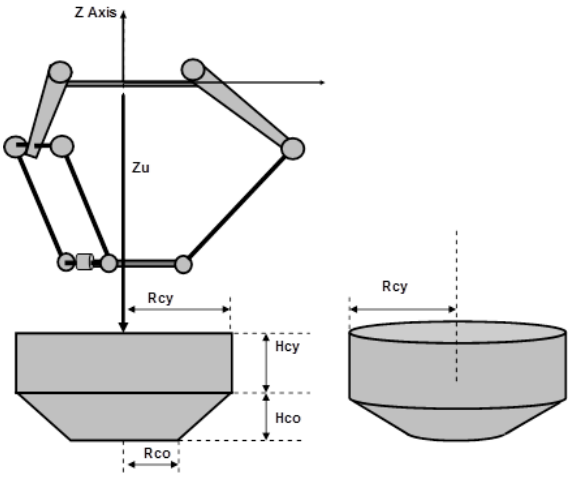


Figure 3.5: The configuration space: a cylinder and a frustum cone

$\mathcal{CS}$, so there are no parts of $\mathcal{CS}$ that are uncovered.

To develop an algorithm for generating valid configurations for two robots, we must first take a look at how this $\mathcal{CS}$ looks like. The $\mathcal{CS}$ of a single robot is a somewhat complex structure. It consists of the intersection of a hexagonal prism with three volumes generated by the three arms of the robot [1]. In order to simplify the configuration space to something more suitable, it is described by a cylinder and a frustum cone (see Figure 3.5). A valid configuration for two robots is thus a point in the cartesian product of two of these configuration spaces.

In order to simplify the procedure to generate samples, we will first generate samples for the configuration space of one robot. After that, we can simply take the cartesian product of two sample sets (for two robots) to represent the entire configuration space. We apply the following

procedure to generate a set of samples (x, y, z) for one robot. Note that the z-axis represents the axis along which the heights of the cylinder and frustum cone are defined (see Figure 3.5). Now, we first fix the z-value for the current sample. We do this by subdividing the z-axis in equally sized intervals, and take the center points of these intervals. Next, we check whether this z-value represents a sample in the cylinder or in the frustum cone. When we fixed the z-value, we reduced the shape in which we sample from a cylinder or frustum cone to a circle. Now, we need to know the radius of this circle to check whether the sample we take is inside or outside the circle. If the z-value represents a sample in the cylinder, then we know already the radius of the circle: this is the same as the radius of the cylinder itself. If the z-value represents a sample in the frustum cone, we need to calculate the radius of the circle.

We can calculate the radius at a particular height in the frustum cone as follows. In Figure 3.6 we depicted a 2D projection of the frustum cone, where we extended the frustum cone to be a 'real' cone. Assume that the largest radius of the cone is r_2 and the smallest radius of the cone is r. The height of the cone, when measured from the largest radius to the smallest radius, is h_r.

Further, we want to calculate the radius r_1 of the cone at a given height h. To do this, we use two steps: compute the angle α and then compute the radius r_1. To compute α, we use the following formula: $\tan(\alpha) = \frac{h_r}{r_2 - r}$. Now, $\alpha = \arctan\left(\frac{h_r}{r_2 - r_1}\right)$. To compute r_1, we use the following derivation.

$$r_1 = \frac{z}{\tan(\alpha)} = \frac{r_2 \cdot \tan(\alpha) - h}{\tan(\alpha)} = r_2 - \frac{h}{\tan(\alpha)}$$

where

$$z = b - h = r_2 \cdot \tan(\alpha) - h$$

Once that we know the z-value and the radius of the circle, we want to obtain for all grid cells the center points that are inside the circle. In Figure 3.7 we depicted a grid that is placed on top of a solid drawn circle. Here, all blue and green points are the points of the grid that we want to include as a sample and the red points are the points that we omit. The blue points can be included by simply looking at whether they are inside the solid drawn circle. Having only these blue points in our sample set is not enough, since there is still some space inside the circle that will not be covered by any of the squares with blue center points. The squares that contain this 'left-over' space have green center points. To include also these green points, we do not check whether any point is inside the solid drawn circle, but whether the point is inside the dashed dotted circle. This latter circle has a radius that is larger than the radius of the solid circle (the difference is the radius of a grid cell). As one can see, the samples we take do not represent exactly a circle (they represent the area enclosed by the fat line in Figure 3.7). Actually, they represent an approximation of the circle, which will be close to a circle when the amount of samples is 'large enough'.

Once that we have all ingredients to sample the configuration space of a single robot, we can describe the algorithm. The algorithm is depicted in Algorithm 1. It follows the same structure as described above: first, we obtain the samples in the z-dimension by using the procedure GET-SAMPLES. This procedure subdivides a dimension into intervals of equal size. This size is computed from the amount of samples k and the length of the largest dimension (line 4).

Every interval will be represented by the point that is in the center of the interval. This center of an interval represents a z-value that we will use in samples. More specifically, for this z-value we compute the circle radius, as described above (using procedure GET-CIRCLE-RADIUS). Using this radius, we subdivide the x- and y-dimension into equally sized intervals. Here every interval is of same length as the intervals that subdivide the z-dimension. We store the center points of these intervals of the x- and y-dimension in the X and Y sets. The cartesian product of these sets represents the set of the two-dimensional points depicted in the grid in Figure 3.7. For each of these points we want to check whether they are inside or outside the circle. For this, we use the

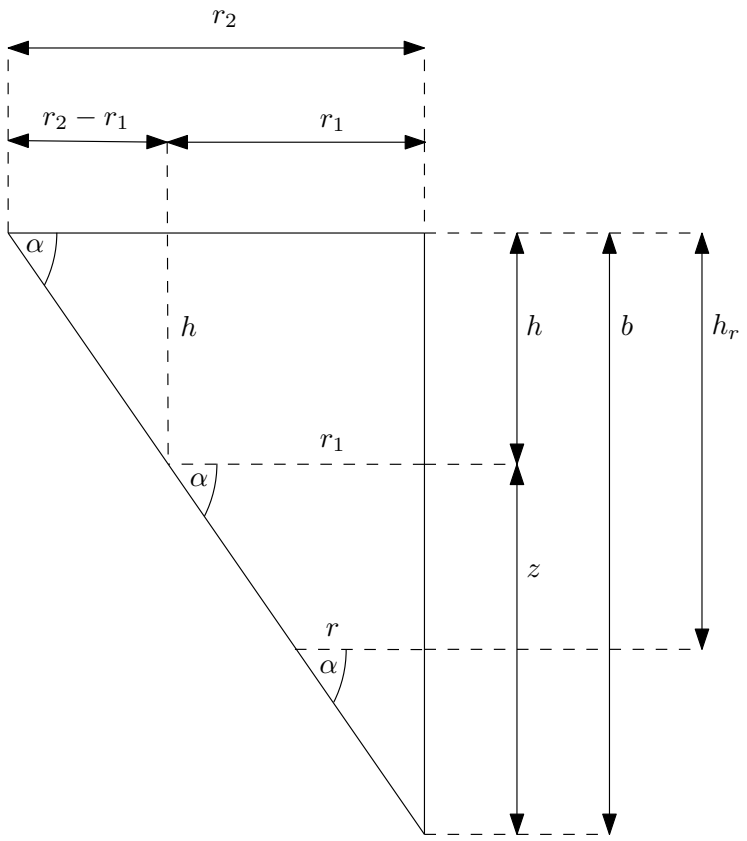


Figure 3.6: Determining the circle radius in the frustum cone

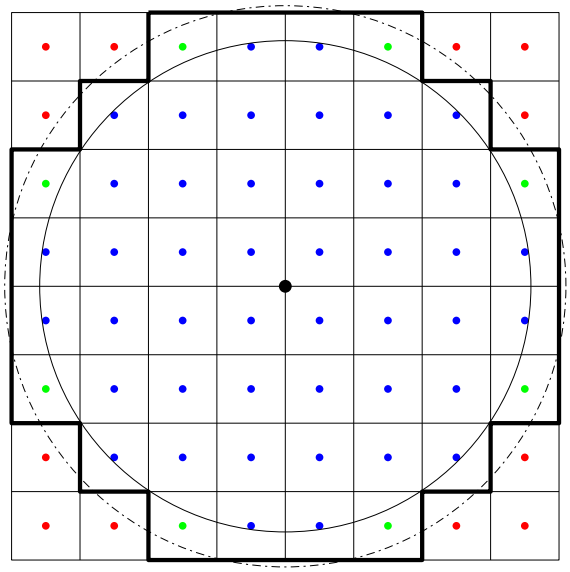


Figure 3.7: Grid placed over a circle from $\mathcal{CS}$ in which red points are omitted in the sample set.

already computed radius of the (dashed dotted) circle and the procedure IS-IN-CIRCLE. Then, for the current z-value, we add the cartesian product of the X and Y sets to the resulting sample set.

Algorithm 1 Generate valid samples for the TCP position of one robot

Input: k, amount of samples in largest dimension, Z_u, H_{cy}, R_{cy}, H_{co} and R_{co}, the height offset, the height and radius of the cylinder and the height and radius of the frustum cone, respectively

function GENERATE-CONFIGURATIONSPACE-SAMPLES($k, Z_u, H_{cy}, R_{cy}, H_{co}, R_{co}$)
  $S \leftarrow \emptyset$
  $\mathcal{CS}_r \leftarrow \max\{R_{cy}, Z_u + H_{cy} + H_{co}\}$
  $r \leftarrow \frac{\mathcal{CS}_r}{k}$
  $Z \leftarrow$ GET-SAMPLES($r, -Z_u - H_{cy} - H_{co}, -Z_u$)
  for $z \in Z$ **do**
    $c_r \leftarrow$ GET-CIRCLE-RADIUS($z, Z_u, H_{cy}, R_{cy}, H_{co}, R_{co}$)
    $X \leftarrow$ GET-SAMPLES($r, -c_r, c_r$)
    $Y \leftarrow$ GET-SAMPLES($r, -c_r, c_r$)
    $T \leftarrow \emptyset$
    for $(x, y) \in X \times Y$ **do**
      if (x, y) is inside the circle with radius $c_r + \frac{r}{\sqrt{2}}$ **then**
        $T \leftarrow T \cup \{(x, y, z)\}$
    $S \leftarrow S \cup T$
  return S
function GET-SAMPLES(r, l, u)
  $S \leftarrow \emptyset$
  $n \leftarrow \lfloor \frac{u-l}{r} \rfloor$
  for $i \leftarrow 1$ to n **do**
    $S \leftarrow S \cup \{i * r + \frac{r}{2} + l\}$
  return S
function GET-CIRCLE-RADIUS($z_i, Z_u, H_{cy}, R_{cy}, H_{co}, R_{co}$)
  if $Z_u \leq z_i \leq Z_u + H_{cy}$ **then**
    return R_{cy}
  else
    $z_i \leftarrow z_i - H_{cy} - Z_u$ ▷ compute the height offset in the frustum cone
    $\lambda \leftarrow \frac{H_{co}}{R_{cy} - R_{co}}$ ▷ as explained above
    return $R_{cy} - \frac{z_i}{\lambda}$ ▷ using the height offset z_i

3.2.3 Sample labeling

After generating a discretization of $\mathcal{CS}$ (as described in Section 3.2.2), we need to compute whether the sampled configurations represent situations in which the robots collide (or not). For this, we use the SAT collision detection algorithm, see Section 3.1. For every configuration $c = (x, y, z, x', y', z')$, we construct the two robot models, where the TCP of these models is placed at (x, y, z) and (x', y', z'), respectively. Then we invoke the SAT algorithm on these two robot models, and store whether this configuration results in a collision or not in the corresponding sample.

As stated earlier, $\mathcal{CS}$ is a six-dimensional space, which means that if we want to place k samples in every dimension, we will end up with $O(6^k)$ samples in total. Because of this the number of samples can grow really large and the procedure to label them will be costly to execute. To address this issue, we developed a parallelized procedure to exploit the multi-core power of modern processors. This procedure invokes SAT on a different configuration for each thread simultaneously. We performed an experiment to see how much we could gain from this multi-core power. For this we generated 35,354,916 samples and constructed for each sample the two corresponding robot models. Then, for every sample, we invoked the SAT algorithm. The code for this experiment is written in C++, compiled *without* optimization and run on a laptop with Windows 7 64 bits

SP1 on a Core i7 2860QM 2.50GHz processor with 8GB of memory. The results are depicted in Figure 3.8. As one can see, the improvement is about a four time speedup, which has mainly practical advantages.

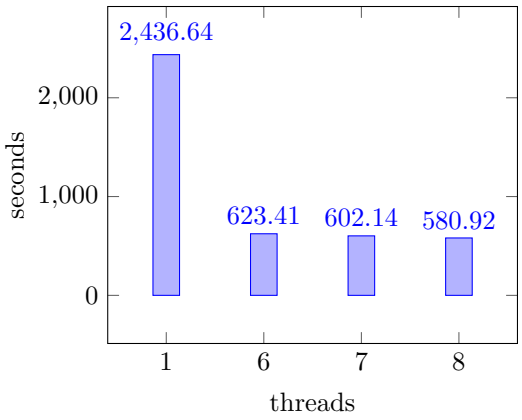


Figure 3.8: Comparision of the performance of labeling samples with a different amount of threads.

Another, more interesting, approach would be to perform adaptive sampling. With adaptive sampling one tries to look, by means of some heuristic, more at 'interesting' parts and less at 'uninteresting' parts of the configuration space. In the current context this means that we want to sample more in the areas where collisions occurs, so that we know more precisely where 'exactly' they occur. The uninteresting parts are the areas in which there is no collision at all.

One way to exploit this is to look at situations in which the minimum distance between two robots is 'large'. If it is large, a small movement of the robot, which corresponds to nearby samples, will not introduce a collision.

A first step could be to relate a movement in the configuration space (the distance between two configurations) to the physical movement of the robot. This would probably involve integrating the direct or inverse kinematics engine. Then, using some coarse approximation of the robot model, for example by using an axis-aligned bounding box, one can compute the movement this bounding volume will make with respect to a movement in the configuration space. If the movement of the bounding volume is related to the minimal distance between two robot models, one could argue in some cases that this minimal distance is too large for a particular movement of the coarse bounding volume (the axis-aligned bounding box). This movement can be translated to a movement in the configuration space, and therefore parts of the configuration space can be identified in which no collisions can occur.

3.2.4 Linear quadtrees

Once that we have discretized the configuration space, labeled the samples into the categories 'collision' and 'no-collision', we can finally describe the data structure that we use to efficiently store and retrieve them. We will first describe the data structure in summary and then explain why exactly this data structure is suitable to store the samples.

A *quadtree* is a rooted tree in which its nodes represent a square in $\mathbb{R}^2$. Every node has four children, where each child represents a quadrant of the square of its parent node. See Figure 3.9 for an example. Here, a square is subdivided into four quadrants, in which the quadrant at North-East (right top) is further subdivided. This subdivision is related to the information that will be stored in the quadtree. The subdivision usually takes place until the quadrants contain information that is 'simple' enough to be represented in a leaf. In the example, this is the case when there is at most one point in a quadrant. All the quadrants that the leaves in this tree represent form again the original square. In other words, the leaves represent a subdivision of the square of the root of

the tree. Further, the *height* of a quadtree is the maximum number of nodes one can encounter when going from the root to any leaf (including the leaf). This definition will be important later on in the description of various algorithms on the quadtree. Similarly, the *depth* of a certain leaf (or node) ν is the number of nodes one encounters when going from the root to ν.

The advantage of using a quadtree is that it stores only information in the areas that are actually subdivided. Referring to the example, the quadtree does not store extra nodes in the North-West quadrants, since there is no further subdivision there. For a more detailed introduction to quadtrees and its properties, we refer to [3, 7].

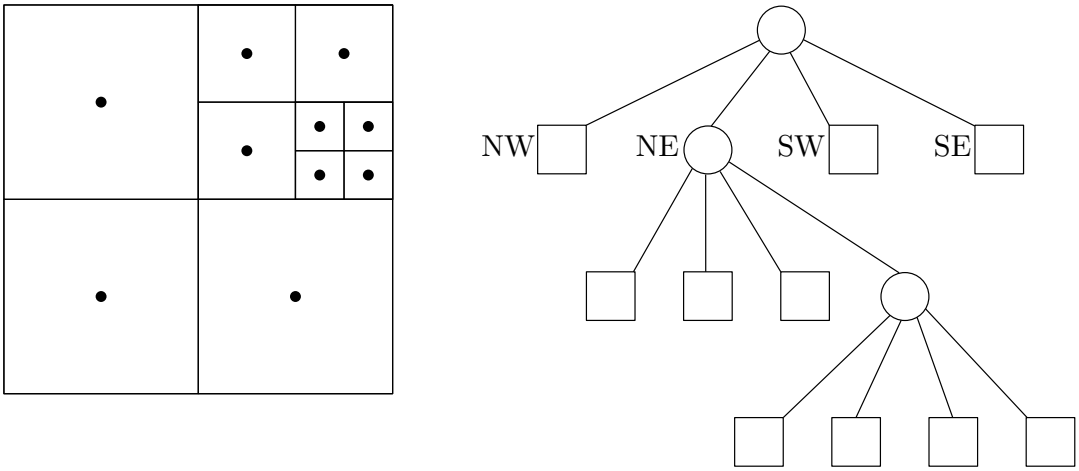


Figure 3.9: A quadtree and the corresponding subdivision of a square.

There are two reasons to use a quadtree. First, the set of samples that we took from the configuration space has a particular structure. The samples are taken from a deterministic grid that has a fixed distance between neighbouring samples. This is a spatial subdivision and this sampling grid can be aligned with the subdivision of the quadtree. Secondly, we expect samples that result in a collision to be 'grouped'. This grouping means that the samples that result in a collision are not distributed like random noise over the configuration space. Instead, we expect them to be contained in manifolds. When storing these samples in the quadtree, they will end up in the same branches of the quadtree. This behavior can be exploited to further reduce the amount of space needed to store the quadtree, as explained in Section 3.2.4.

Formally, a quadtree stores information about a two-dimensional subdivision. Since the samples have six dimensions, we extend the quadtree to be able to handle the four extra dimensions. Every node in the quadtree now represents a 6D hypercube and contains $2^6 = 64$ children (instead of representing a square and storing four children). Further every 6D hypercube should contain only one sample. With a slight abuse of notation, we refer to this 'extended quadtree' simply as quadtree throughout this thesis. To be able to explain some concepts central to a quadtree easily, we explain these concepts in the case of a two-dimensional quadtree. Where needed, we explain how to extend this to the six-dimensional case.

One of the limitations of this approach is the amount of memory that is used. The entire quadtree must fit into primary memory, as explained in Section 1.3. With a quadtree, many unnecessary pointers (and nodes) are stored. For every node, 64 pointers for the children are stored. At the leaves, however, we store only one boolean value, indicating whether there is a collision, or not. Keeping track of all 64 pointers seems thus to be inefficient. All nodes and pointers are basically to guide the search in the quadtree. To get rid of this extra information, we use a particular variant of the quadtree: the *linear quadtree* [4]. In this variant of the quadtree one only stores the leaves of a 'traditional' quadtree in a sorted array. We refer to this sorted

array as the linear quadtree. Since there are no nodes left to guide the search from the root to a particular leaf, the leaves must be indexed in some other way. This is done using *location codes*. A location code contains information that represents the search that would be done in an equivalent 'traditional' quadtree. It encodes which nodes would have been visited. An example of this is given in Figure 3.10. Here, quadrants of a square are numbered by 0, 1, 2 or 3 (corresponding to North-West, North-East, South-West and South-East, respectively). A quadrant of some square (that can be a quadrant itself) is subdivided in the same way, and the current *index* of the quadrant is added to the location code. The index of a quadrant is a number representation to distinguish between all quadrants at the current level. In this case the indices would be 0, 1, 2 and 3.

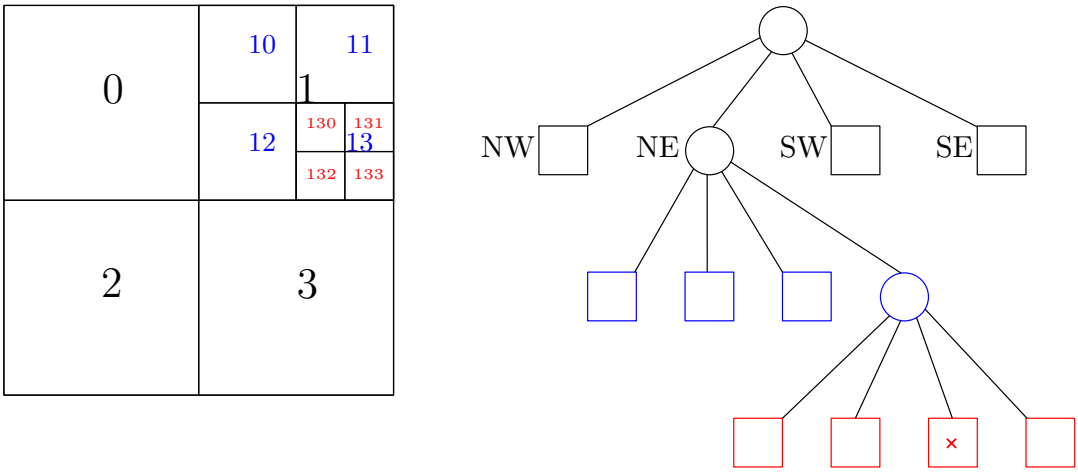


Figure 3.10: A quadtree and the corresponding subdivision indexed by location codes.

Location codes

As suggested in Figure 3.10 location codes contain multiple *levels*. A level represents directly a level in the hierarchy of the quadtree. For a particular leaf ν, the number of levels needed to describe it is thus equal to the depth of ν. For every node that one would encounter when going from the root to ν a level in the location code is filled with the corresponding index of the quadrant.

The index of the quadrant at a particular level is usually represented compactly by the amount of bits necessary to encode one level. In the case of a linear two-dimensional quadtree, two bits are required: this provides room to store the indices for the $2^2 = 4$ quadrants 0, 1, 2 and 3. These bits are then 'shifted to the left' two places, so that the index of the next level's quadrant can be added. This shifting to the left is the same as multiplying the number with 2^2. We denote a location code a d bits location code when for every level d bits are used to represent the index at that level. Formally, a two bits location code L that represents the indices l_j for h levels, where $j \in \{1, 2, ..., h\}$ is calculated as follows:

$$L = \sum_{j=1}^{h} l_j \cdot 4^{h-j}$$

In the case of the six-dimensional linear quadtree, instead of two bits, six bits are used. The formula stays the same in this case, except that we need to multiply the indices with a larger value (to make room in the current bit representation for the new level). In the general case of having a d-dimensional linear quadtree, location codes are computed by the following formula. A d bits location code L that represents the indices l_j for h levels, where $j \in \{1, 2, ..., h\}$ is calculated as follows:

$$L = \sum_{j=1}^{h} l_j \cdot 2^{d(h-j)}$$

Consider the leaf marked with a cross in Figure 3.10. The corresponding indices for this leaf are $l_1 = 1$, $l_2 = 3$ and $l_3 = 2$. We use a two bits location code:

$$L = 30 \overset{\text{base2}}{=} \underbrace{01}_{\text{index } l_1} \quad \underbrace{11}_{\text{index } l_2} \quad \underbrace{10}_{\text{index } l_3}$$

The procedure to compute location codes of the samples for the linear quadtree is depicted in Algorithm 2. Note that we use seven bits in this procedure to represent one level because one extra bit is used for condensing the linear quadtree, as will be explained next.

Algorithm 2 Generate a location code for the given sample

Input: s, a sample for which we generate a location code and h, the height of the quadtree

1: $L \leftarrow 0$
2: $j \leftarrow 1$
3: **for** j to h **do**
4: $\quad l_j \leftarrow$ index of the child at the current level that s belongs to
5: $\quad L \leftarrow L \ll 7$
6: $\quad L \leftarrow L \mid l_j$

Condensation

As explained earlier, the set of samples has a structure that we can exploit by using quadtrees. One of the expected properties is that the samples that result in collisions are somehow grouped. In the quadtree this will mean that the samples will be grouped in the same internal nodes. There will be internal nodes that do not have any children (no collision samples) and there will be nodes that contain many children, depicting that in the part of the space represented by this internal node there are many collision samples. The internal nodes that do not have any children are already compressed by the linear quadtree: none of the children will be included as a location code. The internal nodes that contain all of their children (and thus collision samples) can be *condensed*. In the current setting all children of these internal nodes will be included as a location code. It is possible, however, to store only the internal node and not its children. The location code of the internal node will then be marked, so that during a search one knows that all children of this node will represent in a collision. The search can then be stopped. This condensation is described in [4].

An example is depicted in Figure 3.11. Here, the North-East node of the root contains all its children (colored in red, blue, orange and green, respectively). Without any condensation, all red, blue, orange and green leaves would be stored as location codes in the linear quadtree. By condensing this linear quadtree, one only stores the location code for the North-West child node of the root (marked with a cross). The location code will then be marked in an extra bit, so that during a search one knows that all the children of the current node will represent collisions. This saves storage of sixteen location codes at the cost of using one extra bit for each level in a location code. This extra bit is the seventh bit we use in Algorithm 2.

An algorithm to condense a linear quadtree is depicted in Algorithm 3. Note that all location codes in the linear quadtree are sorted before the invocation of this algorithm. Further, we denote the linear quadtree with $\mathcal{T}$ and use $\mathcal{T}[i]$ to refer to the i-th location code stored in the linear quadtree.

The algorithm loops over all levels in the location codes (line 2). For every level, it checks for all location codes whether it can find ranges that can be condensed. We denote with L_b the

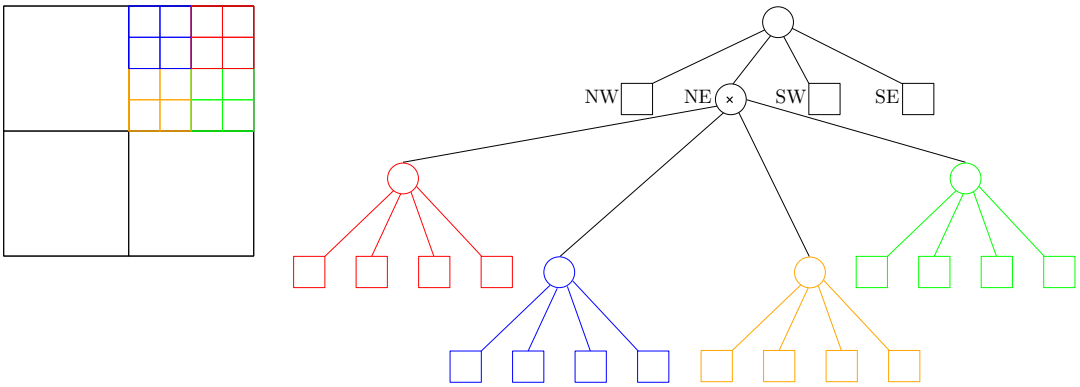


Figure 3.11: A quadtree and the corresponding subdivision indexed by location codes.

location code that is the first in this range. Similarly, we denote with L_e the location code that is at the end of the range. The *prefix* of a location code contains all bits up to a current level. So for a two bits location code at the second level (out of four levels in total), the prefix will consist of the fourth, third and second level. This corresponds to six prefix bits. In a 'traditional' quadtree this would correspond to whether they have the same path from the root to the node at the current level. A range of location codes can be condensed if it satisfies the following constraints:

1. The index of the location code L_b at the beginning of the range must be 0, i.e. it must represent the first child node (line 7).
2. The index of the location code L_e at the end of the range must be $2^d - 1$, i.e. it must represent the last child node (line 9).
3. The prefix of L_b and L_e must match (line 9).

If L_b and L_e satisfy constraints 1 and 2, we have that all location codes in between L_b and L_e of the range must have indices in between the two boundaries. This is because the location codes are sorted. This means that we have a range of location codes in which the indices are in order: $1, 2, 3, ..., 2^d - 1$. If the range satisfies all these conditions, the entire range can be removed from the linear quadtree, except for L_b (line 12). The location code L_b is then used to mark that this entire range has been condensed (the appropriate bit is set, see line 13).

Algorithm 3 Condense a linear quadtree

Input: $\mathcal{T}$, a linear quadtree, n, the amount of leaves in $\mathcal{T}$, h, the height of the linear quadtree and d, the number of bits to represent each level in the linear quadtree

```
1:  i ← 1
2:  for i to h do
3:      j ← 0
4:      for j to n do
5:          c ← FALSE
6:          ν ← T[j]
7:          if index of ν is 0 then
8:              μ ← T[j + 2^d − 1]
9:              if index of μ is 2^d − 1 and all bits up to level i are the same as in ν then
10:                 c ← TRUE
11:         if c then
12:             remove all leaves from T between ν and μ (including μ)
13:             mark ν that it has been condensed
```

Constructing the linear octree

Now that we have described the two main ingredients, we can describe how to construct the linear quadtree. The algorithm is depicted in Algorithm 4. It is given a set of samples, where every sample represents a collision. For each sample, a location code is generated, according to the procedure described in Algorithm 2 (line 3). The location codes are added to the linear quadtree (line 4). Then these location are sorted (line 5). Finally, the linear quadtree is condensed, according to the procedure described in Algorithm 3 (line 6).

The height h of the linear quadtree must be known prior to construct the linear quadtree because it is needed in the procedure to compute the location codes. The height plays also an important role in aligning the subdivision that the linear quadtree infers to the actual sampling grid. The goal is to align these two subdivisions of the configuration space in order to be able to sample in every hypercube that a leaf represents. This is necessary since we construct the linear quadtree only on samples that represent collisions. If, during a search in the linear quadtree, one cannot find the particular leaf we conclude that there is no collision because there was no sample that generated this leaf (more about the search in the query phase in Section 3.2.4). This will only be correct if we have sampled at least once in every hypercube that the subdivision of the linear quadtreee infers.

In the sampling procedure (see Algorithm 1), the resolution of the sampling grid is computed and referred to as $\mathcal{CS}_r$. Further, the radius of the space in which samples are taken is also computed and is referred to as r. The linear quadtree will be constructed on a hypercube with radius r of which its center point is placed at exactly the center point of the configuration space. This to let the hypercube inferred by the linear quadtree and the configuration space overlap. The radius of every hypercube represented by a leaf in the linear quadtree must be equal to $\mathcal{CS}_r$. It will then be equal to the radius of the squares inferred by the sampling grid. By choosing these parameters carefully, we have the guarantee that there is a sample for every hypercube that is contained in the subdivision of the linear quadtree. Samples that do not result in a collision will not end up in the linear quadtree, but there has been one sample for every possible leaf in the linear quadtree.

Algorithm 4 Generate a linear quadtree containing all collision samples

Input: S, a set of collision samples and h, the height of the linear quadtree

1: $\mathcal{T} \leftarrow \emptyset$
2: **for** $s \in S$ **do**
3: $\quad L \leftarrow$ COMPUTE-LOCATION-CODE(s, h)
4: $\quad \mathcal{T} \leftarrow \mathcal{T} \cup \{L\}$
5: SORT($\mathcal{T}$)
6: CONDENSE-TREE($\mathcal{T}$)

The linear quadtree is implemented as the C struct `LinearQuadtree`, defined in Listing 3.1. To analyze the amount of memory that is used by the linear quadtree, we make the following assumptions:

1. A `double` costs eight bytes.
2. A `int` costs four bytes.
3. A `unsigned long long int` costs eight bytes.

Observation 3.2.1. *Assume that the linear quadtree contains m location codes (after condensation). The following expression describes the amount of memory (in bytes) the linear quadtree will cost:*

$$m \cdot cost(\texttt{unsigned long long int}) + 7 \cdot cost(\texttt{double}) + 2 \cdot cost(\texttt{int}) = 8m + 64$$

```
struct Point
{
  double x, y, z, u, v, w;
};

struct Bounds
{
  double radius;
  Point centerPoint;
};

struct LinearQuadtree {
  int numOfNodes;
  unsigned long long int * nodes;
  int numOfLevels;
  Bounds initialBounds;
};
```

Listing 3.1: Linear quadtree definition

Querying the linear quadtree

During system operation, we want to determine from the linear quadtree whether two robots at a particular configuration collide. This means that given a configuration $c = (x, y, z, x', y', z')$, we want to obtain a boolean value that is TRUE when the two robots collide, and FALSE otherwise. The procedure for this is depicted in Algorithm 5.

The procedure is basically a binary search on the (sorted) location codes. The difficulty here is that the location codes stored in the linear quadtree might be condensed. The binary search will then fail in some cases to detect an exact match, i.e. the case $\nu = L$. This is the case because while marking a particular level of a location code that it is condensed (the extra seventh bit), we change the number value of the location code. The location code L that is computed from the configuration c is then not equal to the stored location code. Therefore, we apply a different equality operator. A location code L *equals a condensed node* ν when the following conditions are satisfied (used in lines 12, 14 and 16):

1. For all levels in both L and ν they match (exactly) or,
2. for the level in which ν and L do not match, ν contains the condensation mark.

Another implication of this marking of condensed nodes is that a binary search does not have to end at a node ν that corresponds (under the relaxation for comparing condensed nodes) with the location code L. Since we set the extra seventh bit in ν to mark that it has been condensed, we actually increased its number value. Therefore, at the end of a binary search, the search could stop at a value lower than the corresponding node ν. To solve this, we check both the lower- and upperbound at the end of the binary search (lines 14 till 17).

Theorem 3.2.1. *A linear quadtree that contains six-dimensional collision samples can be constructed such that it costs* $8m + 64$ *bytes, where* m *is the number of location codes stored in the linear quadtree. The linear quadtree can be queried in* $O(\log m)$ *time.*

Proof. The expression about the amount of space used follows from Observation 3.2.1. The query procedure, Algorithm 5, performs a binary search on m location codes. The binary search takes $O(\log m)$ time, but the query procedure performs at every step in the while loop (line 5) an extra comparison: a comparison that takes into account that ν could be condensed. This comparison takes constant time and thus does not influence the complexity of the query.

After the while loop (line 5) we perform two extra comparisons that take into account that $\mathcal{T}[low]$ and $\mathcal{T}[high]$ might be condensed. Again, because these comparisons take constant time, the complexity of the query is not influenced. Hence, the query has the same time complexity of a binary search, which is $O(\log m)$. ■

Algorithm 5 Query a linear quadtree

Input: c, the configuration of the two robots, $\mathcal{T}$, the linear quadtree, h, the height of the linear quadtree and n, the number of location codes in the linear quadtree

```
1: L ← COMPUTE-LOCATION-CODE(c, h)
2: low ← 0
3: high ← n
4: middle ← ⌊n/2⌋
5: while high − low ≥ 2 do
6:     ν ← T[middle]
7:     if ν < L then
8:         low ← middle
9:         middle ← low + (right−low)/2
10:    else if ν > L then
11:        high ← middle
12:        middle ← low + (right−low)/2
13:    else if ν = L then
14:        return TRUE
15:    else if ν is a condensed node that equals L then
16:        return TRUE
17: if T[low] is a condensed node that equals L then
18:    return TRUE
19: else if T[high] is a condensed node that equals L then
20:    return TRUE
21: return FALSE
```

3.2.5 Linear 5D quadtree

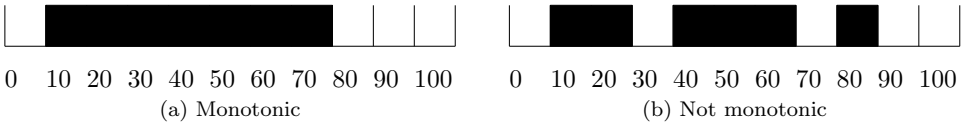
The memory requirements of the linear quadtree can still be high (as shown in Section 3.2.9) and therefore we investigate in this section whether there are additional methods we can use to compress the linear quadtree. The linear quadtree described in Section 3.2.4 contains samples from a six-dimensional configuration space. Therefore, the indexation procedure (the procedure to compute the location codes) and the query procedure takes into account all six dimensions. An interesting question to investigate is: what if we only consider five of the six dimensions from the linear quadtree (and thus adapt also all procedures to ignore this dimension) and store at the leaves of the linear quadtree the 'ignored' dimension?

More specifically, if one dimension is ignored from the collision samples, a linear five-dimensional quadtree (linear 5D quadtree) is constructed on these samples. Then at the leaves of the linear 5D quadtree there can be a collection of samples, that represent this ignored dimension. This approach could be visualized as follows. For a configuration $c = (x, y, z, x', y', z')$, we store the first five coordinates in the linear 5D quadtree, so we store configurations like $c' = (x, y, z, x', y')$. This corresponds with fixing the pose of one robot by fixing the coordinates (x, y, z) and fixing the the pose of the second robot to be on a particular position in the x', y'-plane, but for which the z'-coordinate can vary. Multiple z'-coordinates can exist for this five-dimensional configuration. All these z'-coordinates are then stored at the leaf (an example of this can be the interval depicted in Figure 3.12a and 3.12a. If the collection of samples would have some structure, we could save space in storing these z'-coordinates.

A central assumption we make for this approach (and the approach taken for the linear four-dimensional quadtree) is the following: *The configuration space of two robots is mostly monotonic.* A justification (apart from experimental verification) is this following intuition. When two robots collide at some particular configuration $c = (x, y, z, x', y', z')$ they cannot move further towards each other. For example, they cannot move further in the x- and x'-dimension, because they already collide at the current configuration c and moving further in these dimensions will cause

the robots to move more inside of each other. This does not only hold for the x- and x'-dimension, but for all dimensions (in particular cases). In these situations, the corresponding configurations that represent collisions will then be grouped consecutively in the configuration space.

When the configuration space is monotonic, the one-dimensional problem could look like the situation depicted in Figure 3.12a and not like Figure 3.12b. In these figure, a black box means that there is a collision at the corresponding value, while a white box means that there is no collision. More precisely, an interval is monotonic if it contains only one consecutive subinterval of collision configurations.



(a) Monotonic (b) Not monotonic

Figure 3.12: Interval that is monotonic (left) and an interval that is not monotonic (right).

To exploit this monotonicity, we take the following approach. Let the *collision interval* be the subinterval that contains the collision configurations in the ignored dimension. If the configuration space is truly monotonic, the collision intervals will contain all collision configurations consecutively (or none at all). Now, instead of storing for every configuration in the ignored dimension a location code (as is done with the linear six-dimensional quadtree), we store only the coordinate at which the collision interval begins and the coordinate at which the collision interval ends. In Figure 3.12a, this would mean that we store only the coordinates 10 and 80, instead of the range 10, 20, ..., 80.

Since it is not guaranteed that the configuration space is entirely monotonic, we need to implement support for non-monotonic situations. To provide support for this is actually not too hard. We can store for every subinterval of configurations the begin and end coordinate.

Constructing the linear 5D quadtree

In Algorithm 6 we describe how to construct this linear 5D quadtree. Without loss of generality, we will assume (throughout the thesis) that we ignore the z' dimension from configuration $c = (x, y, z, x', y', z')$.

The algorithm expects as input a sorted list of six-dimensional collision samples. The ordering on the collision samples is lexicographic. Further, the algorithm loops over this list S and performs operations over ranges in which the first five coordinates are equal (line 2). Hence, the range contains samples that only differ on the ignored dimension z'. Then, the corresponding location code is computed (line 5). This location code is computed only on five dimensions (excluding the ignored dimension) and uses only five bits for every level. We use five bits per level since we do not condense the linear 5D quadtree (hence we do not need the extra bit for this). We do not condense the linear 5D quadtree, because while marking a node as condensed, the 64 children might have different collision intervals. One could merge these collision intervals, but then information will get lost.

Since the five coordinates for the entire current range are the same, the location code will be equal for all samples in this range. Hence, it does not matter which sample from the current range is picked to compute the location code for. Next, we compute the begin and end coordinates for all intervals in this range (line 6). Note that if the ignored dimension is truly monotonic, we will have only one interval here. This list of subintervals is then stored together with the five-dimensional location code in the linear quadtree (line 7). Finally, the nodes consisting of the location codes and list of subintervals in the linear 5D quadtree are sorted on the location codes.

The linear 5D quadtree is implemented as the C struct `Linear5DQuadtree`, defined in Listing 3.2.

Algorithm 6 Generate a linear 5D quadtree containing all collision samples

Input: S, a sorted list of collision samples and h, the height of the linear 5D quadtree

1: $\mathcal{T} \leftarrow \emptyset$
2: **for** every 5D range in S **do**
3: $S_I \leftarrow$ all samples that belong to the current 5D range
4: $s \leftarrow$ one of the samples in S_I
5: $L \leftarrow$ COMPUTE-LOCATION-CODE(s, h)
6: $I \leftarrow$ the begin and end coordinates of all intervals found in S_I
7: $\mathcal{T} \leftarrow \mathcal{T} \cup \{(L, I)\}$
8: SORT($\mathcal{T}$)

Observation 3.2.2. *Assume that the linear 5D quadtree contains m location codes together with i intervals, counted over all location codes. The following expression describes the amount of memory (in bytes) the linear 5D quadtree will cost:*

$$m{\cdot}(cost(\texttt{unsigned long long int})+cost(\texttt{int}))+i{\cdot}(2cost(\texttt{double}))+7{\cdot}cost(\texttt{double})+2{\cdot}cost(\texttt{int}) = 12m+16i+64$$

```
struct Point
{
  double x, y, z, u, v, w;
};

struct Bounds
{
  double radius;
  Point centerPoint;
};

struct Interval {
  double begin;
  double end;
};

struct Node5DInt {
  unsigned long long int locationCode;
  int numOfIntervals;
  Interval * intervals;
};

struct Linear5DOctree {
  int numOfNodes;
  Node5DInt * nodes;
  int numOfLevels;
  Bounds initialBounds;

  int numOfZValues;
  double * zValues;
};
```

Listing 3.2: Linear 5D quadtree definition

Querying the linear 5D quadtree

The procedure to query the linear 5D quadtree is the same as for the six-dimensional variant, except for two main differences. The first is that since the linear 5D quadtree is not condensed, we do not have to perform a custom equality check on condensed leaves. Secondly, in addition to the binary search we have to 'solve' the one-dimensional query problem. This one-dimensional

problem can be solved as follows. Given the z' coordinate of the ignored dimension, there is a collision if there is a interval that contains this z' coordinate.

The procedure is depicted in Algorithm 7. Here we denote the location code of a node ν in the linear 5D quadtree as $L_c(\nu)$. Further, we denote the set of intervals stored with a node ν as $\mathcal{I}(\nu)$. The begin and end coordinate of an interval I is denoted as $b(I)$ and $e(I)$. Now, the lines 1—13 are a basic binary search on the location codes of the nodes in the linear 5D quadtree. When a match on the location codes has been found, the z' coordinate of the configuration has to be checked. To do this, we loop over all intervals stored at the current node ($\mathcal{I}(\nu)$) and for every interval we check whether it contains the z' coordinate. If this is the case we report that there is a collision (line 17), otherwise we report that there is no collision (line 18).

Algorithm 7 Query a linear 5D quadtree

Input: $c = (x, y, z, x', y', z')$, the configuration of the two robots, $\mathcal{T}$, the linear 5D quadtree, h, the height of the linear 5D quadtree and n, the number of location codes in the linear 5D quadtree

```
1:  L ← COMPUTE-LOCATION-CODE(c, h)
2:  low ← 0
3:  high ← n
4:  middle ← ⌊n/2⌋
5:  found ← FALSE
6:  while high − low ≥ 2 do
7:      ν ← T[middle]
8:      if L_c(ν) < L then
9:          low ← middle
10:         middle ← low + (right−low)/2
11:     else if L_c(ν) > L then
12:         high ← middle
13:         middle ← low + (right−low)/2
14:     else if L_c(ν) = L then
15:         found ← TRUE
16: if L_c(ν) = L ∨ found then
17:     for I ∈ I(ν) do
18:         if b(I) ≤ z' ≤ e(I) then
19:             return TRUE
20: return FALSE
```

Theorem 3.2.2. *A linear 5D quadtree can be constructed such that it costs* $12m + 16i + 64$ *bytes, where* m *is the number of location codes stored in the linear 5D quadtree and* i *is the number of intervals over all location codes that is stored in the linear 5D quadtree. The linear 5D quadtree can be queried in* $O(\log m + i_m)$ *time, where* i_m *represent the maximum number of intervals stored at any location code.*

Proof. The expression about the amount of space used follows from Observation 3.2.2. Now we have to prove that the linear 5D quadtree can be queried in $O(\log m + i_m)$ time. The proof follows from the fact that a binary search is performed over m elements, which results in a time complexity of $O(\log m)$. In addition to this we have to take into account that after the binary search we check for all intervals at the node we found whether they contain the current z' coordinate. The containment check can be done in constant time and is repeated at most i_m times. Hence the time complexity of the query is $O(\log m + i_m)$. ■

3.2.6 Linear 4D quadtree

The linear 4D quadtree takes the approach from the linear 5D quadtree one step further. In the linear 4D quadtree we ignore two dimensions instead of one. Instead of storing intervals at the leaves of the linear 4D quadtree, we now store 2D slices. These slices are the result of the (cartesian) product of two intervals, one interval for each ignored dimension. A 2D slice is thus a grid on which the squares can be black: the configuration results in a collision, or white: the configuration does not result in a collision. In Figure 3.13 an example of such a 2D slice is depicted.

We again would like to make the assumption that the configuration space is mostly monotonic, which would allow us to compress the 2D slices. If this would not be the case, compressing the 2D slices could be hard. For example, when the 2D slices would contain a random configuration of black and white squares. The monotonicity property does not help explicitly here (in contrast to the linear 5D quadtree), but helps implicitly.

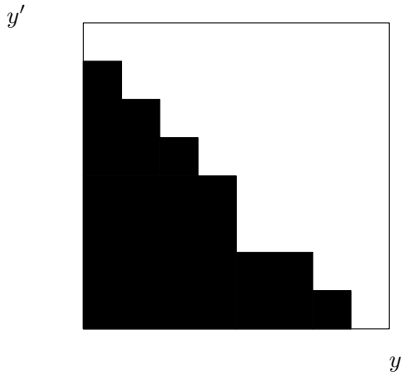


Figure 3.13: An example of how a 2D slice looks like. The black squares represent collision configurations and the white squares represent non collision configurations.

To get an idea whether this assumption is realistic, we did an experimental evaluation. We generated all 2D slices from the configuration spaces in which the two corresponding robots are placed at different distances and angles from each other and using different parameters for the robot model (D_a and H_a to be precise). We could verify that most 2D slices are monotonic and the 2D slices that were not strictly monotonic looked mostly monotonic (there were a few configurations that 'broke' this monotonicity property). The Figures 3.16 and 3.17 show *quilts* of 2D slices, that we considered in our experimental evaluation. A quilt is basically a grid of squares where at each square the corresponding 2D slice is placed. For example, we sort the samples in the following order x, x', z, z', y, y'. The quilts will be constructed for every pair (x, x') and the quilts themselves will represent the z and z' dimension. Then, at a fixed square in the quilt, a 2D slice consisting of the dimensions y and y' is depicted.

This visualization of the configuration space tries to capture what patterns occur in a 4D part of the configuration space. At the left and bottom side of the quilt the coordinates of the grid on the quilt have been placed. In Figure 3.16 the grid of the quilt is placed over the zz'-slice of the configuration space. Then, at every square in the quilt, 2D yy'-slices are placed. In Figure 3.17 the grid of the quilt is placed over the yy'-slice of the configuration space and at every square in the quilt, 2D zz'-slices are placed.

Studying these figures there are (at least) three approaches one could take to compress the 2D slices:

1. Combine similar 2D slices.
2. Store the 2D slices more efficiently by interpreting the collision areas as geometric shapes.
3. Apply hashing techniques to find patterns/similar 2D slices.

In this thesis, we will implement approach 1: 'Combine similar 2D slices'. The idea is to not store the 2D slices at the leaves of the linear 4D quadtree, but instead to store a pointer to the 2D slice. Leaves that have similar slices will then store a pointer to the same 2D slice, so that the 2D slice has to be stored only once. The problem with this approach is how to define 'similarity' between two slices, as this problem is now basically a clustering problem.

Similarity

We define *similarity* between two slices with the help of a certain threshold t and the *difference* between two slices. The difference between two slices is computed as the cardinality of their symmetric difference. The symmetric difference of two sets A and B in general is defined as:

$$A \triangle B = (A \setminus B) \cup (B \setminus A)$$

To be able to apply this definition to 2D slices, we define a 2D slice $\mathcal{S}$ as being a rectangular arrangement of squares, in which the squares represent collision configurations. Further, we split the definition of the symmetric difference for two slices $\mathcal{S}_1$ and $\mathcal{S}_2$ in two parts. The first part, $\mathcal{S}_1 \triangle_{add} \mathcal{S}_2$ is the set of squares that is present in $\mathcal{S}_1$, but is not in $\mathcal{S}_2$, i.e. $\mathcal{S}_1 \setminus \mathcal{S}_2$. The second part, $\mathcal{S}_1 \triangle_{sub} \mathcal{S}_2$ is the set of squares that is not present in $\mathcal{S}_1$, but is present in $\mathcal{S}_2$, i.e. $\mathcal{S}_2 \setminus \mathcal{S}_1$. With a slight abuse of notation, we will denote the difference $\triangle_{add}$ between two slices $\mathcal{S}_1$ and $\mathcal{S}_2$ as $\triangle_{add}(\mathcal{S}_1, \mathcal{S}_2)$ and similarly for the difference $\triangle_{sub}$ as $\triangle_{sub}(\mathcal{S}_1, \mathcal{S}_2)$. Finally, we denote the cardinality of these sets using the operator: $|\cdot|$.

Having defined the difference between two slices, we can describe how we compute the similarity between two slices. If $t = 0$, we say that the two slices $\mathcal{S}_1$ and $\mathcal{S}_2$ are similar if there is an exact match. This means that for every square in $\mathcal{S}_1$ and $\mathcal{S}_2$ they must agree on whether it is black or white. Formally, this is the case when

$$|\triangle_{add}(\mathcal{S}_1, \mathcal{S}_2)| = 0 \wedge |\triangle_{sub}(\mathcal{S}_1, \mathcal{S}_2)| = 0$$

If we allow t to become larger, the two slices do not have to match exactly anymore. We split the threshold t into two other thresholds $t_{add} = t$ and $t_{sub} = t$. These two thresholds are initialized with the value of t, but during the execution of Algorithm 8 the bookkeeping for both thresholds is different. Intuitively, the two slices match when there need to be at most t_{add} squares colored black in $\mathcal{S}_1$ to match with $\mathcal{S}_2$ and there need to be at most t_{sub} squares colored white in $\mathcal{S}_1$ to match with $\mathcal{S}_2$. Formally, two slices match when:

$$|\triangle_{add}(\mathcal{S}_1, \mathcal{S}_2)| < t_{add} \wedge |\triangle_{sub}(\mathcal{S}_1, \mathcal{S}_2)| < t_{sub}$$

An example is given in Figure 3.14. Here, the slices $\mathcal{S}_1$ and $\mathcal{S}_2$ are as depicted in Figure 3.14a and their difference is depicted in Figure 3.14b. The $\triangle_{add}(\mathcal{S}_1, \mathcal{S}_2)$-difference is depicted as gray squares and the $\triangle_{sub}(\mathcal{S}_1, \mathcal{S}_2)$-difference is depicted as the square with dashed dotted border. If we would set $t = 0$, then there is no match between the two slices. Also for $t = 1$ there is no match, since $|\triangle_{add}(\mathcal{S}_1, \mathcal{S}_2)| = 2$. For $t > 1$ the slices match.

The assumed monotonicity property will indirectly help with grouping slices because of the following. When having this property and the particular shape of the slices (rectangles with missing corners), we hope that similar slices are only a few bits different from each other. More specifically, these bits cannot be in the entire space of the slice, but would rather be close to the boundary. Since the union of these slices defines a new boundary, other similar slices can 'more easily' merge with them. More easily because they would look similar to this new boundary, as opposed when having 'random noise' slices.

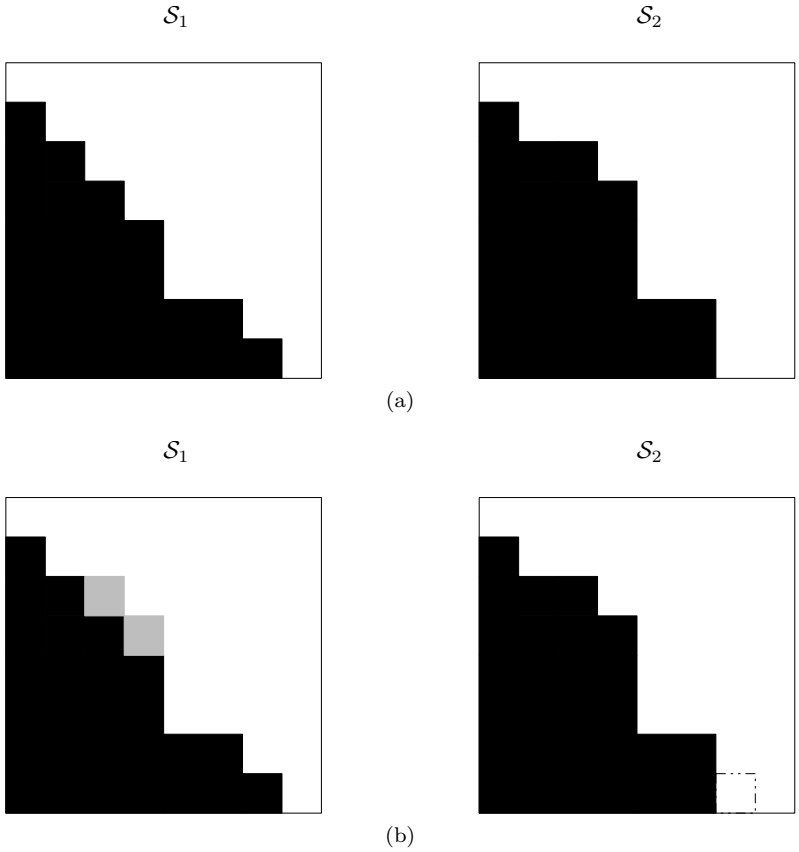


Figure 3.14: Two 2D slices $\mathcal{S}_1$ and $\mathcal{S}_2$ that are matched to each other (3.14a). The add-differences are depicted on the left in 3.14b and the sub-differences are depicted on the right.

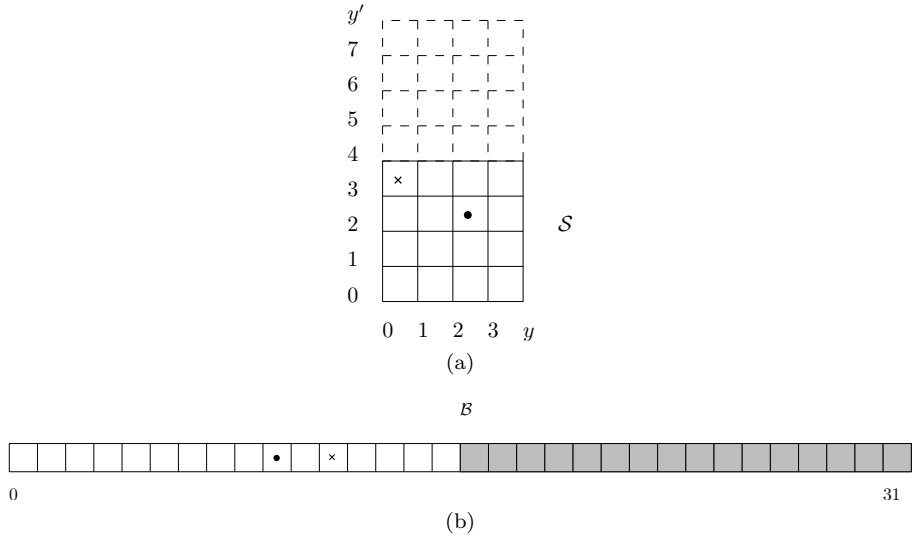


Figure 3.15: A 2D slices $\mathcal{S}$ and the corresponding bit array $\mathcal{B}$).

Constructing the linear 4D quadtree

Having defined a notion of similarity between two slices, we can describe the exact representation of the slices. The representation is important because of two reasons. The first is that it could affect the time complexity of the query phase. The second is that it directly affects the space requirements of the linear 4D quadtree. To store a slice, we use *bit arrays*. A bit array is a list of integers of which the bits can be addressed independently. The reason to use a bit array is twofold. First, the time to test a specific bit is constant, hence a particular configuration can be tested for collision or non collision in constant time (as will be explained in 3.2.3). The second reason is that the amount of memory used by this representation is not tremendously large. Better approaches could exist, but the focus of the current approach is not on having the representation the minimal number of bits. Instead, the focus is more on whether 2D slices can be grouped together.

The amount of bits in a bit array is always a multiple of the amount of bits used to represent an integer. Recall that in Section 3.2.4 we assume an integer to be represented by four bytes and thus 32 bits. To represent a slice of d^2 squares (the sides have length d) we need $\lceil \frac{d^2}{32} \rceil$ integers in a bit array. The bit array addresses its bits in a linear range, so we have to create a mapping between the squares in a slice and the bits in a bit array.[1] For a slice with sides of length d a position (i, j) (counted from left bottom, see Figure 3.15a) is represented at index $i * d + j$ in the bit array.

In Figure 3.15b a bit array $\mathcal{B}$ consisting of one integer is depicted. In Figure 3.15a a yy'-slice $\mathcal{S}$ is depicted. The slice $\mathcal{S}$ consists of sides with length four, hence it contains sixteen squares. The squares in Figure 3.15a that have dashed borders are the squares for which there is space in the bit array, but that are not used. These places are colored gray in $\mathcal{B}$. To compute the indices of the positions in $\mathcal{B}$ that correspond with the square containing the disk S_D and the square containing the cross S_C, we use the mapping described earlier. The position of S_D is $4 * 2 + 2 = 10$ and the position of S_C is $4 * 3 + 0 = 12$.

Next, we describe the algorithm to construct the linear 4D quadtree. Without loss of generality, we will sort the samples lexicographically in the order x, x', z, z', y, y' and thus the 2D slices stored at the leaves of the linear 4D quadtree correspond to the yy'-slices. The procedure is depicted

[1] Actually, it addresses its bits in chunks of 32 bits so that they are distributed over the various integers. We will not explain this here since it does not contribute to the discussion of using bit arrays.

in Algorithm 8. It follows the same structure of Algorithm 4 and Algorithm 6, but it includes some operations to construct, store and index the 2D slices. The algorithm will construct two structures: $\mathcal{T}$, the linear 4D quadtree and $\mathcal{G}$, the list that contains all 2D slices indexable by an integer. More specifically, $\mathcal{G}$ contains groups G that consist of a 'representative slice' $\mathcal{S}_G$, and two thresholds G_{add} and G_{sub}. Both $\mathcal{T}$ and $\mathcal{G}$ are initialized empty (line 1 and 2). Then, the algorithm loops over all 4D ranges from the sample set S (line 3). It constructs the 2D slice from the current samples $\mathcal{S}$ (line 4) and starts to compare this with all currently stored 2D slices (line 7). The comparison is done against the representative $\mathcal{S}_G$ of the current group G. If they are similar: the $\triangle_{add}$-difference and the $\triangle_{sub}$-difference are both below the current thresholds G_{add} and G_{sub}, respectively, the index of G is stored (line 14). Further, the union of $\mathcal{S}$ and $\mathcal{S}_G$ is computed and stored in G (line 11 and 12). The representative of this group G is computed by the union of the previous representative $\mathcal{S}_G$ and the current slice $\mathcal{S}$, because we only want to introduce extra collision configurations, instead of possibly removing them. If, for example, a collision configuration would be removed, it is possible that while querying $\mathcal{T}$ one ends up at exactly this bit and reports that there is no collision (since we did not include this configuration), while actually there is a collision. Since we compute for every match the union of the two similar slices, the representative in G gradually 'grows': collision configurations will be added. To avoid that more than t_{add} bits will be added throughout the entire run of the algorithm, a threshold G_{add} will be maintained together with the representative. If a new slice is matched, this threshold G_{add} will be lowered by the $\triangle_{add}$-difference. At some point, the threshold G_{add} will be zero or small enough so that new slices cannot be added (and merged) to the current group G.

If no similar slice is found (line 16), the current slice $\mathcal{S}$ is added to $\mathcal{G}$ and the corresponding index is stored (line 17–19). Finally, the location code is computed and stored together with the index in the linear 4D quadtree (lines 20–23). As a final step, all location codes are sorted (line 24).

The linear 4D quadtree is implemented as the C struct `Linear4DQuadtree`, defined in Listing 3.3.

Observation 3.2.3. *Assume that the linear 4D quadtree contains m location codes together with s slices which contain b^2 squares each. The following expression describes the amount of memory (in bytes) the linear 4D quadtree will cost:*

$$m \cdot cost(\texttt{unsigned long long int}) + s \cdot \frac{b^2}{8} + b\,cost(\texttt{double}) + 7\,cost(\texttt{double}) + 5\,cost(\texttt{int}) = 8m + \frac{sb^2}{8} + 8b + 76$$

```
struct Node4DInt {
  unsigned long long int locationCode;
  unsigned int sliceIndex;
};

struct Slice {
  int* bitArray;
};

struct Linear4DOctree {
  int numOfNodes;
  Node4DInt * nodes;
  int numOfLevels;
  Bounds initialBounds;

  int numOfSlice;
  int numOfIntsInSlice;
  Slice* slices;
  int numOfDimValues;
  double * dimValues;
};
```

Listing 3.3: Linear 4D quadtree definition

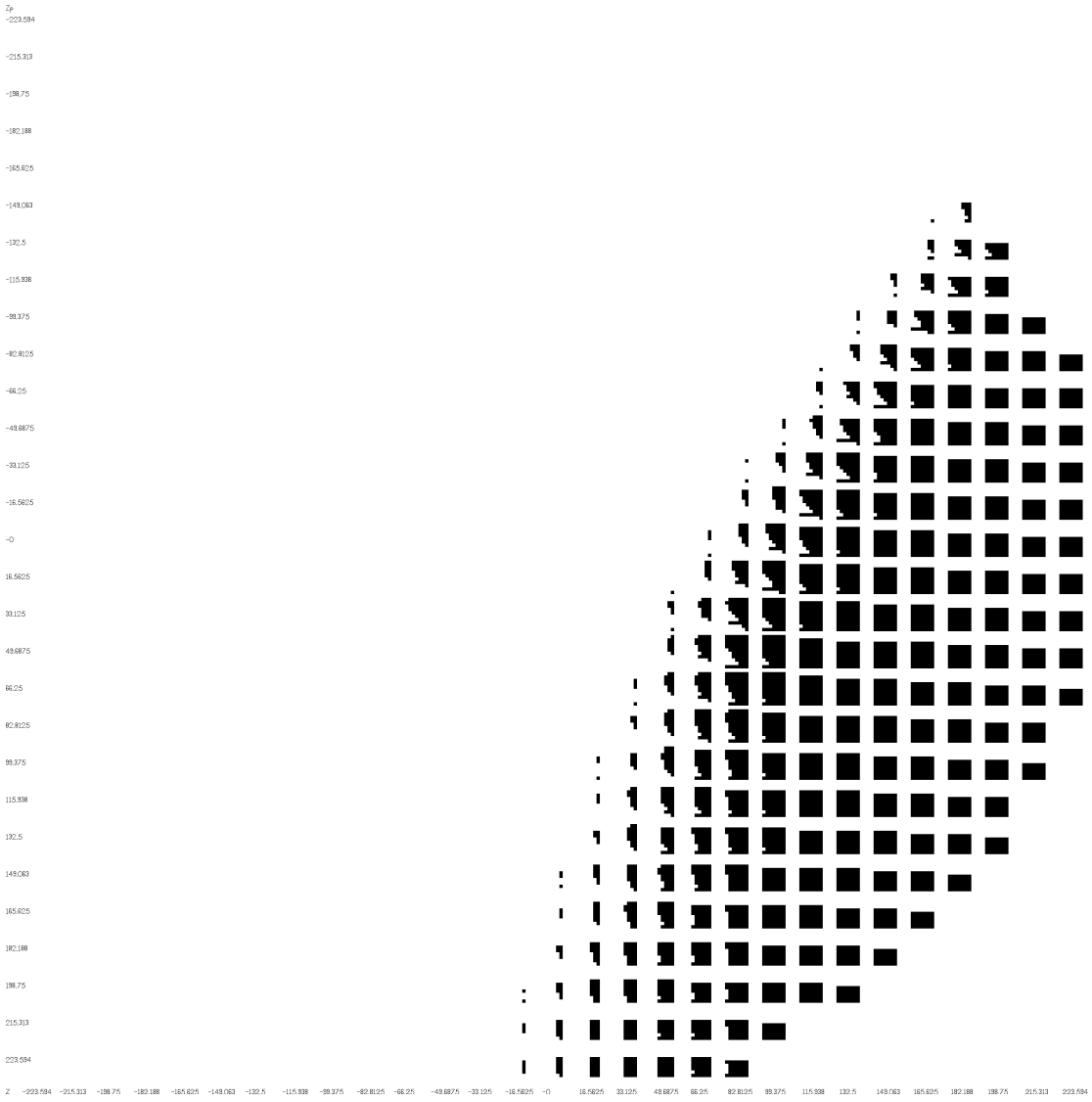


Figure 3.16: A quilt with yy'-slices of the configuration space. Black boxes represent collision configurations and white boxes represent no-collision configurations.

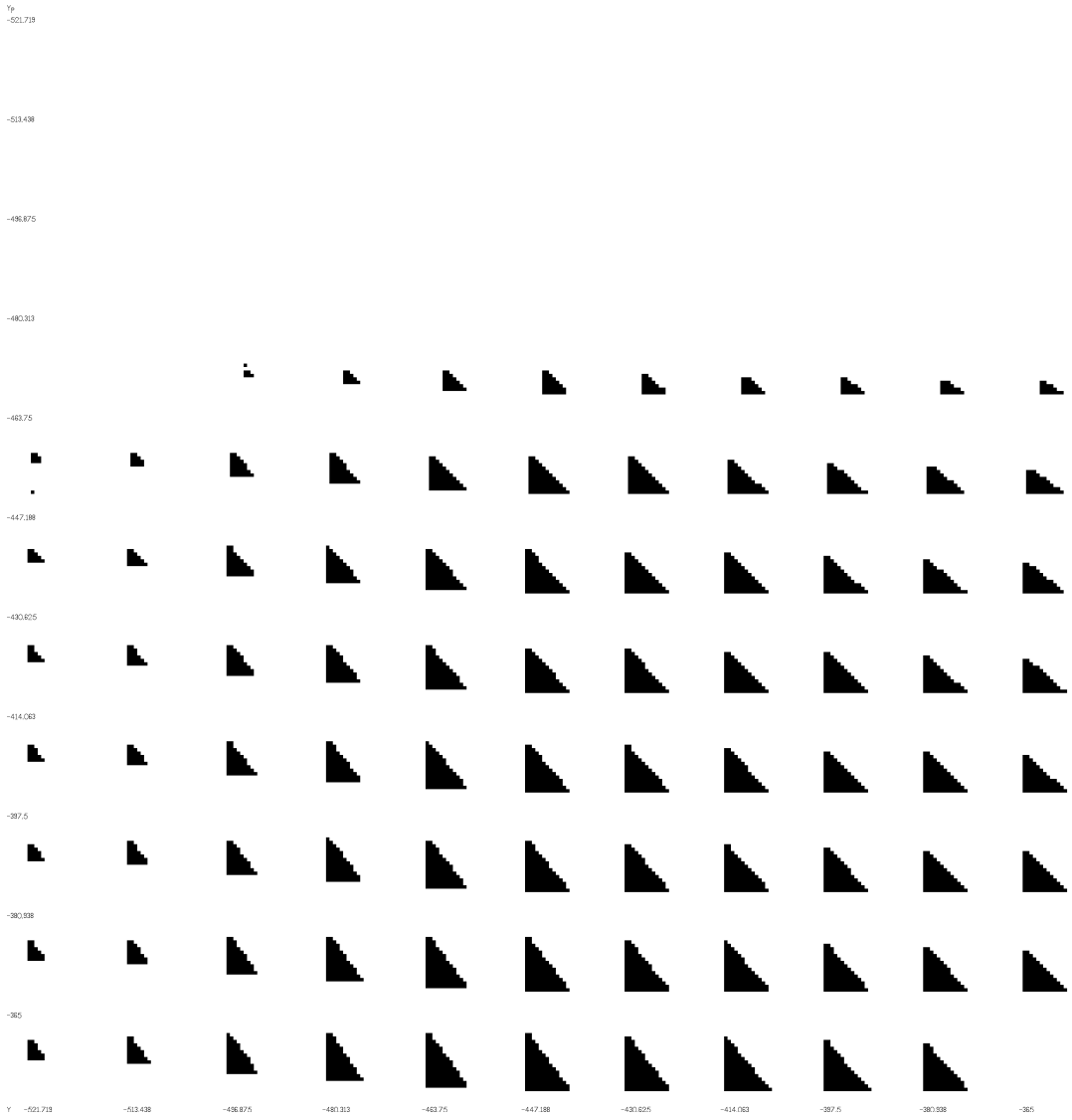


Figure 3.17: A quilt with zz'-slices of the configuration space. Black boxes represent collision configurations and white boxes represent no-collision configurations.

Algorithm 8 Generate a linear 4D quadtree containing all collision samples

Input: S, a sorted list of collision samples, h, the height of the linear 4D quadtree, t_{add} and t_{sub}, the thresholds for adding and subtracting squares, respectively

1: $\mathcal{T} \leftarrow \emptyset$
2: $\mathcal{G} \leftarrow \emptyset$
3: **for** every 4D range in S **do**
4: $\quad\mathcal{S} \leftarrow$ the 2D slice corresponding to the current 4D range
5: $\quad index \leftarrow -1$
6: $\quad found \leftarrow$ FALSE
7: $\quad$**for** $(i, \mathcal{S}_G, G_{add}, G_{sub}) \in \mathcal{G}$ **do**
8: $\quad\quad\tau_{add} \leftarrow \triangle_{add}(\mathcal{S}, \mathcal{S}_G)$
9: $\quad\quad\tau_{sub} \leftarrow \triangle_{sub}(\mathcal{S}, \mathcal{S}_G)$
10: $\quad\quad$**if** $\tau_{add} < G_{add} \wedge \tau_{sub} < G_{sub}$ **then**
11: $\quad\quad\quad\mathcal{S}_U \leftarrow$ the union of $\mathcal{S}$ and $\mathcal{S}_G$
12: $\quad\quad\quad\mathcal{S}_G \leftarrow \mathcal{S}_U$
13: $\quad\quad\quad G_{add} \leftarrow G_{add} - \tau_{add}$
14: $\quad\quad\quad index \leftarrow i$
15: $\quad\quad\quad found \leftarrow$ TRUE
16: $\quad$**if** $\neg found$ **then**
17: $\quad\quad i \leftarrow$ the highest index in $\mathcal{G}$ plus one
18: $\quad\quad\mathcal{G} \leftarrow \mathcal{G} \cup \{(i, \mathcal{S}, t_{add}, t_{sub})\}$
19: $\quad\quad index \leftarrow i$
20: $\quad S_I \leftarrow$ all samples that belong to the current 4D range
21: $\quad s \leftarrow$ one of the samples in S_I
22: $\quad L \leftarrow$ COMPUTE-LOCATION-CODE(s, h)
23: $\quad\mathcal{T} \leftarrow \mathcal{T} \cup \{(L, index)\}$
24: SORT($\mathcal{T}$)

Querying the linear 4D quadtree

The procedure to query the linear 4D quadtree is the same as for the five-dimensional variant, except for checking the structures stored at the leaves. First, a binary search is performed to find the correct node in the linear 4D quadtree. Next, the corresponding slice is retrieved and the bit corresponding to the 2D configuration (y, y') is tested.

The procedure is depicted in Algorithm 9. Here we denote the location code of a node ν in the linear 4D quadtree as $L_c(\nu)$. Further, we denote the slice index stored with a node ν as $I(\nu)$ and the slice in the storage $\mathcal{G}$ at index i as $\mathcal{G}[i]$. Now, the lines 1—13 are a basic binary search on the location codes of the nodes in the linear 4D quadtree. When a match on the location codes has been found, the yy' position in the slice has to be checked. To do this, we retrieve the slice according to the slice index stored at the current node ν by $\mathcal{G}[I(\nu)]$. Then we test whether the bit at the corresponding position has been set. If this is the case we report that there is a collision (line 18), otherwise we report that there is no collision (line 18 and 19).

Theorem 3.2.3. *A linear 4D quadtree can be constructed such that it costs* $8m + \frac{sb^2}{8} + 8b + 76$ *bytes, where* m *is the number of location codes stored in the linear 4D quadtree,* s *is the number of slices and where each slice contains* b^2 *squares. The linear 4D quadtree can be queried in* $O(\log m)$ *time.*

Proof. The expression about the amount of space used follows from Observation 3.2.3. The time complexity of a query for the linear 4D quadtree is $O(\log m)$ since we apply a binary search and at the end we retrieve the indices of the y- and y'-coordinates in constant time (using a map). Retrieving the bit from the bit array takes also constant time. ■

Algorithm 9 Query a linear 4D quadtree

Input: $c = (x, y, z, x', y', z')$, the configuration of the two robots, $\mathcal{T}$, the linear 4D quadtree, h, the height of the linear 4D quadtree, n, the number of location codes in the linear 4D quadtree, $\mathcal{G}$, the list of 2D slices, $\mathcal{D}$, the coordinates along the extracted dimensions and d, the number of coordinates along one extracted dimension in $\mathcal{D}$

1: $L \leftarrow$ COMPUTE-LOCATION-CODE(c, h)
2: $low \leftarrow 0$
3: $high \leftarrow n$
4: $middle \leftarrow \lfloor \frac{n}{2} \rfloor$
5: $found \leftarrow$ FALSE
6: **while** $high - low \geq 2$ **do**
7: $\quad \nu \leftarrow \mathcal{T}[middle]$
8: $\quad$ **if** $L_c(\nu) < L$ **then**
9: $\quad\quad low \leftarrow middle$
10: $\quad\quad middle \leftarrow low + \frac{(right-low)}{2}$
11: $\quad$ **else if** $L_c(\nu) > L$ **then**
12: $\quad\quad high \leftarrow middle$
13: $\quad\quad middle \leftarrow low + \frac{(right-low)}{2}$
14: $\quad$ **else if** $L_c(\nu) = L$ **then**
15: $\quad\quad found \leftarrow$ TRUE
16: **if** $L_c(\nu) = L \vee found$ **then**
17: $\quad \mathcal{S} \leftarrow \mathcal{G}[I(\nu)]$
18: $\quad y_{index} \leftarrow$ the index of y in $\mathcal{D}$
19: $\quad y'_{index} \leftarrow$ the index of y' in $\mathcal{D}$
20: $\quad$ **return** test bit $y_{index} * d + y'_{index}$ in $\mathcal{S}$
21: **return** FALSE

3.2.7 Guarantees

All of the previous approaches with linear quadtrees suffer from one problem. They provide information about the configuration space (collision or no collision) at fixed points—the points that are sampled by the sampling procedure. If the linear quadtree is queried with a configuration c that is not exactly one of these points that are sampled by the sampling procedure, some error occurs. This situation is depicted in Figure 3.18. In this figure, the discs represent the samples that were taken from the configuration space and that result in collisions, hence they are contained in the linear quadtree. The cross, marked with c, is a configuration for which we want to know whether the two corresponding robots are in collision or not. When executing the query for this configuration, it is assigned to the quadrant represented by q and the linear quadtree is queried with q. The problem here is that the configuration at q represents a collision, while it could be the case that c is actually a configuration that does not represent a collision (c is contained in a part of the configuration space where there are no collisions, see the dashed dotted enclosed area in Figure 3.18). This situation will be called a *false positive*, i.e. we falsely report that there is a collision (positive), while there is none. The opposite, reporting that there is no collision, while in fact there is a collision is called a *false negative*. The two other possibilities are that we report correctly whether there is a collision or not, a *true positive* or a *true negative*, respectively.

In the context of this thesis, it must absolutely be avoided to have the possibility to report false negatives. If it would be possible for this to occur, robots could physically collide. In order to eliminate these situations the error has to be reduced. One way of doing this is to 'blow up' the robot model. If the robot model is extended in every dimension by the error the discretization of the configuration space makes, there can never be a possibility for a false negative.

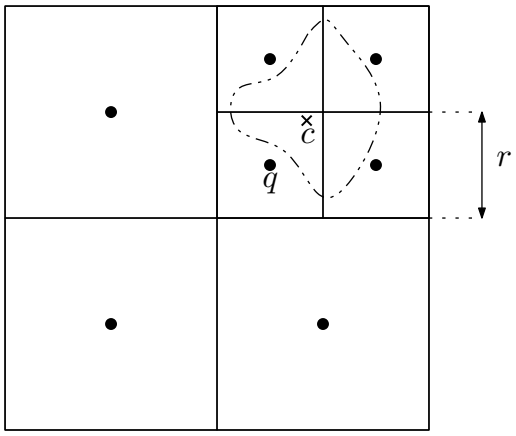


Figure 3.18: A configuration c that is being treated as point q. The dashed dotted enclosed area is a collision free area.

Assume that the diameter of a square in the sampling grid is r (the resolution). In the worst case, configuration c would be in the corner of a square, as is already depicted in Figure 3.18. The distance between the center of the square, q and configuration c is in this worst case scenario is bounded by $\sqrt{\frac{r^2}{2}}$. If the robot model is then, in all dimensions, increased by this size it will eliminate all possibilities for these false negatives.

3.2.8 Symmetries

In order to further reduce the amount of space required by the linear quadtrees, one could look at symmetries. If, for example, two different configurations would be equivalent to each other, we could simply store only the result of one of the configurations. Whether symmetric robot configurations exist depends very much on how the robots are placed with respect to each other. In this section we analyze the symmetries that arise in one of the most common placements for Delta3-R robots, depicted in Figure 3.19. In this setup, two robots, R_1 and R_2 are depicted. The robots contain three arms each (see Section 2.2) of which the upper links are denoted by a_1, a_2 and a_3 for R_1 and by b_1, b_2 and b_3 for R_2. The circles, centered at c_1 and c_2 for robot R_1 and R_2 respectively, denote the reachable area for the robots (the reachable positions of the TCP).[2]. Further, the origin of the coordinate system in which the positions of the robot's TCPs are described is placed at c_1 and the base vectors x and y are aligned as depicted in the picture. Next, the arms of the robots are rotated 120° respectively from each other (around c_1 or c_2).

The symmetry that can be exploited here is a reflection in axis s. When R_1 places its TCP at the box and R_2 places its TCP at the small circle, they will collide if and only if they collide also when R_1 places its TCP at the square and R_2 places its TCP at the disk. More formally, when we have the following configuration of the two robots, $c = (\square_x, \square_y, z, \bigcirc_x, \bigcirc_y, z')$ this will be a collision configuration if and only if the configuration $c' = (\square_x, -\square_y, z, \bigcirc_x, -\bigcirc_y, z')$ also is a collision configuration. The configuration c' is depicted in the figure as $c' = (\blacksquare_x, \blacksquare_y, z, \bullet_x, \bullet_y, z')$.

Lemma 3.2.1. *A mapping from configuration $c = (x, y, z, x', y', z')$, in which $y \geq 0 \wedge y' \geq 0$, to a configuration $c' = (x, -y, z, x', -y', z')$ can be created such that both d and d' represents a collision configuration or none represent a collision configuration.*

Proof. The pose of a_1 for the two TCP positions (x, y, z) and $(x, -y, z)$ (box and square), is equivalent under reflection of axis s. This because the distance from the arm a_1 to the box or

[2] Actually, the reachable positions are not contained in a circle, but in a more complex structure (see Section 3.2.2) For the discussion in this section, we will assume that the reachable positions are enclosed by the circles

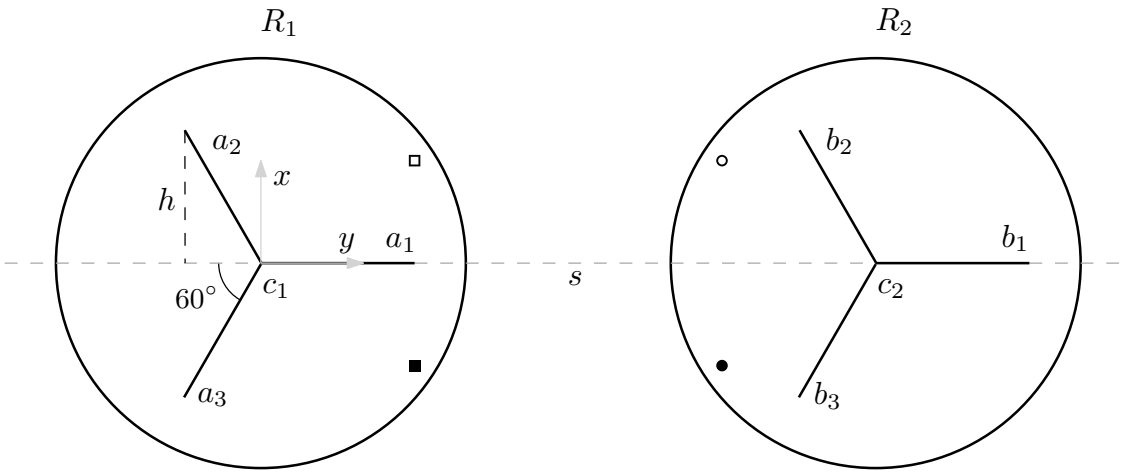


Figure 3.19: A xy-projection of two Delta3-R robots.

square is the same and they are at the same x position. Further, the axis s is an angle bisector of the angle between the arms b_2 and b_3. The pose of arm b_2 and b_3 for the two TCP positions (x', y', z') and $(x', -y', z')$ (circle and disk), is equivalent under reflection of axis a. This is because the distance from b_2 to the disk is the same as the distance from b_3 to the circle, and vice versa. Further, the x position of both the disk and circle are the same. Since the arms b_2 and b_3 are exactly the same they will have equivalent poses under the reflection of axis s.

Having that the pose of the corresponding arms is the same, we can argue the same about whether the particular configurations must both represent collisions or no collisions. The distance between the arms of R_1 and R_2 is the same for both configurations. Since both the distance and the pose (under reflection) are the same, the configurations must agree upon whether they represent a collision or not. ■

Lemma 3.2.2. *The reflection axis s is the only symmetry that can be applied when two Delta3-R robots are placed in a setup as depicted in Figure 3.19.*

Proof. The reflection axis s is given by the *dihedral group* $\mathcal{D}_1$. The dihedral group $\mathcal{D}_1$ has order two and includes the reflection axis s and the cyclic group $\mathcal{C}_1$, which represents a rotation of 360° (i.e. the identity rotation). Having this reflection axis s, one can wonder whether there is another reflection axis, or maybe a rotation that can be used to create another mapping.

An extension to this can be made by looking at whether we can apply dihedral group $\mathcal{D}_2$ here. This group includes another reflection axis s_2 and the cyclic group $\mathcal{C}_2$. The cyclic group $\mathcal{C}_2$ contains two rotations, the identity rotation and the rotation around 180°. This rotation, however, does not map the setup to itself (i.e. is a symmetric rotation), because it will result in the setup that is depicted in Figure 3.20. Therefore, cyclic group $\mathcal{C}_2$ is not applicable here and since dihedral group $\mathcal{D}_2$ contains this cyclic group, $\mathcal{D}_2$ is also not applicable here.

■

Theorem 3.2.4. *When two Delta3-R robots are placed in a setup as depicted in Figure 3.19, the amount of collision configurations that is stored in linear quadtrees can be reduced by a factor two. No other symmetries can be exploited in this setup.*

Proof. From Lemma 3.2.1 it follows that we have a mapping from all positive y and y' coordinates to their negative counterparts (and vice versa). Therefore, we can omit storing half of the dimensions y and y' in the linear quadtrees. More precisely, we can omit storing all configurations $e = (x, y, z, x', y', z')$ where $y < 0 \wedge y' < 0$. During the query phase, we can use the mapping from Lemma 3.2.1 to map configurations e to their positive counterpart. That no other symmetries can be exploited follows directly from Lemma 3.2.2. ■

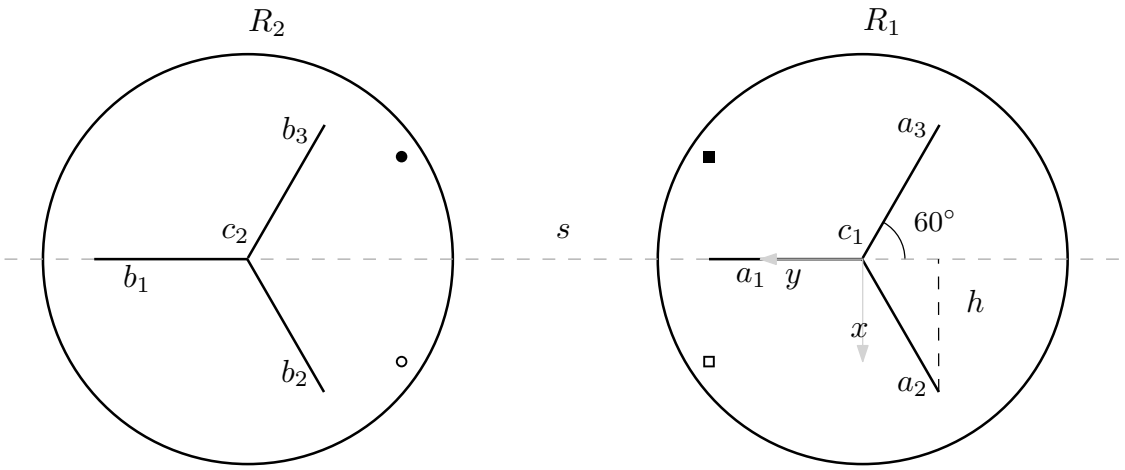


Figure 3.20: A xy-projection of two Delta3-R robots, rotated 180° around c_1.

3.2.9 Performance evaluation

In this section we present results regarding performance, memory requirements and accuracy of the different linear quadtree variants. We conclude this section with a summary.

To gather data about the performance of the linear quadtree approaches, we created three test cases: T_1, T_2 and T_3. Each test case contains a trajectory for each robot in which the TCPs of the robots follow these trajectories. The robots were not able to move along the entire trajectory since the last parts of the trajectories represent configurations in which the robots collide and were stopped (the trajectories let the robots move towards each other). The trajectories that were generated for these movements are 'linear', i.e. the TCPs will move on a line segment through physical space ($\mathcal{W}$). The robots were placed as follows. The first robot R_1, was placed at the origin of the WCS. The second robot R_2 was placed at the position $(465.36, 182.9, 0)$, defined in the WCS. The three test cases have the begin and end points for the TCPs of R_1 and R_2 as depicted in Table 3.1. The trajectories were computed with an internal primitive called MoveTimeAbsolute.

Testcase	**Begin position R_1**	**End position R_1**	**Begin position R_2**	**End position R_2**
T_1	$(0, 0, -380)$	$(215, 0, -400)$	$(0, 0, -380)$	$(-215, 0, -400)$
T_2	$(0, 0, -380)$	$(190, 80, -400)$	$(0, 0, -380)$	$(-190, -80, -400)$
T_3	$(0, 0, -380)$	$(170, 130, -400)$	$(0, 0, -380)$	$(-170, -130, -400)$

Table 3.1: The three test cases with begin and end position for the TCP of R_1 and R_2. The positions are defined in the MCS of the corresponding robot.

All test cases were executed on the NJ-controller, *outside the robotics software system* and time measurements were done by measuring the amount of clock cycles a particular query needs. The amount of clock cycles can be measured by the ClockCycles() method implemented in the QNX OS. The tests were done outside the robotics software system because of complications while gathering statistics about the experiments. The experiments were run, however, on exactly the same hardware, so all performance data is still valid. The code that implements Algorithm 5, 7 and 9 was written in (ANSI) C, compiled with GNU GCC 4.3.3 for a x86 Little Endian architecture under optimization level O2 and warning level w1. The corresponding settings for executing the code on the NJ-controller were that the task in which the code is running has priority 150 and debug information is stripped. The scheduling algorithm used to schedule which tasks can be executed on the NJ-controller had been set to Round Robin.

The configuration space parameters $Z_u, H_{cy}, R_{cy}, H_{co}$ and R_{co} were set to 365, 135, 225, 30 and 173, respectively. These represent the offset, height and radius of the cylinder and the height and (lower) radius of the frustum cone, respectively (see Section 3.5 for details). The units are in millimeters (and correspond directly to the world, $\mathcal{W}$). The parameter k, used to generate the samples and which is linked to the amount of levels in location codes, is defined at each test case separately. The linear quadtree is always constructed with condensation. Note that we take, as explained in Section 3.2.5 the z' dimension as ignored dimension for the linear 5D quadtree. For the linear 4D quadtree, we will define this at each test case separately. Recall that t refers to the threshold for matching similar slices in the linear 4D quadtree (see Section 3.2.6).

The code to generate the linear quadtrees was written in C++, compiled *without* optimization and ran on a laptop with Windows 7 64 bits SP1 on a Core i7 2860QM 2.50GHz processor with 8GB of memory.

Memory requirements

In this section we present the memory requirements of the different linear quadtree approaches. Recall that the parameter k, used as input for the sampling procedure denotes the amount of samples in the largest dimension. In Table 3.8 we depicted the settings that are used for this entire section, unless otherwise stated. The linear 4D quadtree here was generated on yy'-slices, where each slice contains $29^2 = 841$ configurations.

k	**Samples**	**Collision samples**	D_a	H_a
32	35,354,916	584,416	26	79

Table 3.2: Parameters for the memory and time experiments.

In Table 3.3, we show the results of the generated linear quadtrees. The amount of memory was already considerably lower for the linear 5D quadtree than the linear quadtree. The linear 4D quadtree requires much less memory compared to the linear 5D quadtree. The reason is that less location codes had to be stored (requiring 64 bits each). Further, the amount of memory required to store the ignored dimension(s) is not as much as storing all location codes (without the ignored dimension(s)). The reason why the number of slices dropped (when increasing t), is that because of the bigger threshold more slices matched. They were assigned more easily to an existing group, hence less groups were needed for all slices. While increasing t, the amount of memory and slices was decreasing more slowly. For example, increasing t from 100 to 150 resulted in a decrease of 31 slices and 5332 bytes of memory. Increasing t from 50 to 100 resulted in a decrease of 133 slices and 22876 bytes. The amount of memory decrease was directly connected with the decrease in the amount of slices. The number of location codes stored stayed the same, and therefore only the amount of slices stored was of influence on the memory requirements of the linear 4D quadtree. The speed in which the number of slices decreased can be explained by the following argument. For low values of t, similar slices that have a difference 'just' bigger than t were grouped in different slices. While increasing t these slices will be grouped together. For big(ger) values of t the slices that were grouped separately were 'really' different slices and thus a relatively large increase in t was needed to group them together.

In Table 3.4, we show the linear quadtrees for the same situation, but we generated zz'-slices (instead of yy'-slices), where each slice contained $11^2 = 121$ configurations. The behavior was the same as we see with the yy'-slices in Table 3.3, expect for one thing. While increasing t, the number of slices became actually 1. This can be explained by relating t to the amount of configurations that were contained in a slice. If t is 125 or 150, it can group any slice to a single group, since 125 (or 150) bits can be added to any slice. A slice in this situation only contains 121 bits and it is thus possible to group an empty and full slice together.

Data Structure	Time (sec)	#Location codes	Memory (bytes)	#Slices (groups)
6D	7.25	466,984	3,735,936	-
5D	86.26	85,539	2,053,100	-
4D / t = 0	1,312.66	8,548	1,245,460	6,444
4D / t = 5	482.51	8,548	535,272	2,315
4D / t = 10	282.9	8,548	364,476	1,322
4D / t = 25	107.37	8,548	220,340	484
4D / t = 50	51.39	8,548	169,772	190
4D / t = 75	34.37	8,548	154,464	101
4D / t = 100	26.82	8,548	146,896	57
4D / t = 125	23.83	8,548	143,800	39
4D / t = 150	23.16	8,548	141,564	26

Table 3.3: Results for the linear quadtree (6D), linear 5D quadtree (5D), linear 4D quadtree (4D) with yy'-slices. Time refers to the construction time of the data structure. The memory requirements are computed as stated in Observation 3.2.1, 3.2.2 and 3.2.3.

Data Structure	Time (sec)	#Location codes	Memory (bytes)	#Slices (groups)
6D	7.25	466,984	3,735,936	-
5D	86.26	85,539	2,053,100	-
4D / t = 0	70.27	8,548	380,884	2,011
4D / t = 5	19.27	8,548	242,244	278
4D / t = 10	12.39	8,548	229,444	118
4D / t = 25	15.36	8,548	222,164	27
4D / t = 50	13	8,548	220,644	8
4D / t = 75	15.41	8,548	220,244	3
4D / t = 100	12.73	8,548	220,164	2
4D / t = 125	9.23	8,548	220,084	1
4D / t = 150	10.65	8,548	220,084	1

Table 3.4: Results for the linear quadtree (6D), linear 5D quadtree (5D), linear 4D quadtree (4D) with zz'-slices. Time refers to the construction time of the data structure. The memory requirements are computed as stated in Observation 3.2.1, 3.2.2 and 3.2.3.

In Figure 3.21, the memory requirements for the linear 4D quadtree are plotted for both the approach with yy'-slices and with zz'-slices and in Figure 3.22, the number of slices (groups) are plotted. These figures are visualization of the data in Table 3.3 and 3.4. From these figures it seems that the approach to use yy'-slices for the linear 4D quadtree is more successful: it reduced the memory requirements the most.

In Table 3.5, we constructed the linear quadtrees for $k = 64$ (corresponding to six levels in the location codes), sampled 1,807,270,144 configuration and obtained 22,583,625 collision configurations. The linear 4D quadtree here was generated on yy'-slices, where each slice contained $55^2 = 3025$ configurations. The behavior was similar to in Table 3.3 and 3.4. We included a testcase for $t = 545$ here since the slices are much larger, 3025 bits versus 841 bits. When $t = 150$, we allow about 17.8% of a slice (when using slices of 841 bits) to be different. When $t = 545$, we also allows about 17.8% of a slice (when using slices of 3025 bits) to be different.

In Figure 3.23 the memory requirements for the linear quadtres with $k = 64$ are plotted. Here one can see the reduction in memory requirements for the linear 5D quadtree and the linear 4D quadtree, compared with the linear quadtree. Note that for the linear 4D quadtree, $t = 150$ and $t = 545$, so the linear 4D quadtree did not contain exactly the same amount of information as the linear quadtree and the linear 5D quadtree did.

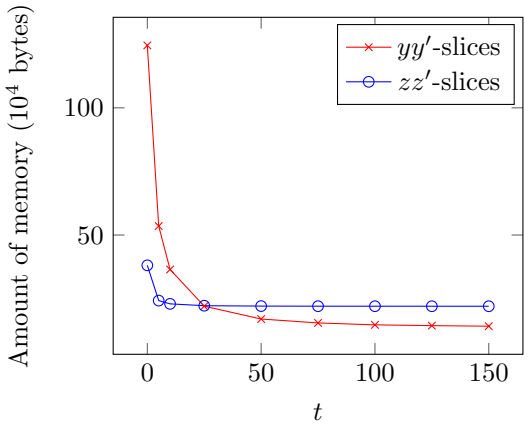


Figure 3.21: The amount of memory (10^4 bytes) of the linear 4D quadtree when using yy'- or zz'-slices.

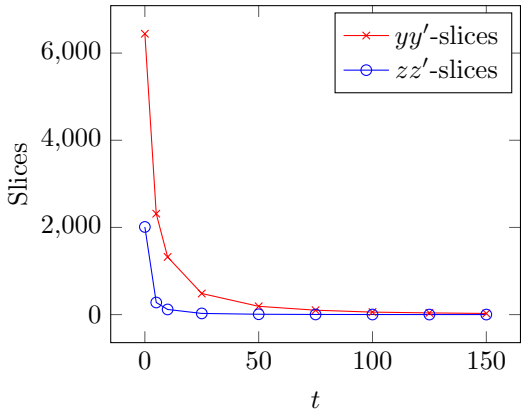


Figure 3.22: The amount slices (groups) of the linear 4D quadtree when using yy'- or zz'-slices.

Data Structure	Time (sec)	#Location codes	Memory (bytes)	#Slices (groups)
6D	4,190.69	11,435,334	91,482,736	-
5D	2,117.36	1,957,285	46,975,084	-
4D / t = 150	5,784.97	104,976	1,969,636	652
4D / t = 545	1,508.01	104,976	1,701,904	49

Table 3.5: Results for the linear quadtree (6D), linear 5D quadtree (5D), linear 4D quadtree (4D), when $k = 64$. Time refers to the construction time of the data structure. The memory requirements are computed as stated in Observation 3.2.1, 3.2.2 and 3.2.3.

Time performance

In this section we present results about the query performance. The results are depicted in Table 3.6 and 3.7. The results were measured by executing the entire trajectories, which consist of 6000 steps. The performance was thus tested over 6000 consecutive queries and the average timings were reported. The resolution of our timing method was around one microsecond. From the results one can conclude that using a linear 5D quadtree or a linear 4D quadtree might (very) slightly improve the performance, but this is almost negligible. The advantage of this is that it seems that the extra time spent on solving the one-dimensional or two-dimensional problems does not exceed the time needed to do a binary search on a larger set of location codes. For example,

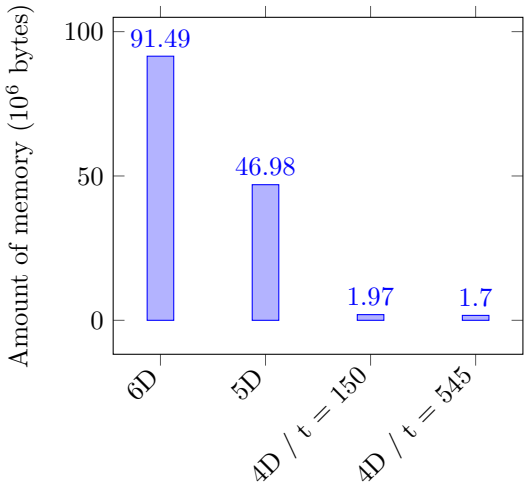


Figure 3.23: The amount of memory (10^6 bytes) of the linear quadtree (6D), linear 5D quadtree (5D), linear 4D quadtree (4D) when using yy'-slices and $k = 64$.

in Table 3.6, one can see that the linear 5D quadtree and linear 4D quadtree were actually a bit faster than the linear quadtree. The reason is that the binary search performed by the linear quadtree had to be applied to a much bigger set of location codes: 466,984 for the linear quadtree, 85,539 for the linear 5D quadtree and 8,548 for the linear 4D quadtree.

Testcase	Linear quadtree	Linear 5D quadtree	Linear 4D quadtree
T_1	3	2	2
T_2	3	2	2
T_3	3	2	2

Table 3.6: Average query time in microseconds for $k = 32$ (584,416 collision configurations). The trajectories of all test cases consist of 6000 positions and thus 6000 queries are performed.

Testcase	Linear quadtree	Linear 5D quadtree	Linear 4D quadtree
T_1	3	2	3
T_2	3	3	3
T_3	3	3	3

Table 3.7: Average query time in microseconds for $k = 64$ (22,583,625 collision configurations). The trajectories of all test cases consist of 6000 positions and thus 6000 queries are performed.

In Figure 3.24, the query timings for the SAT algorithm (including constructing the two robot models) and the linear quadtree variants are depicted. This plot makes it clear that the linear quadtree approach was much faster, mainly because of the cost to compute the robot model.

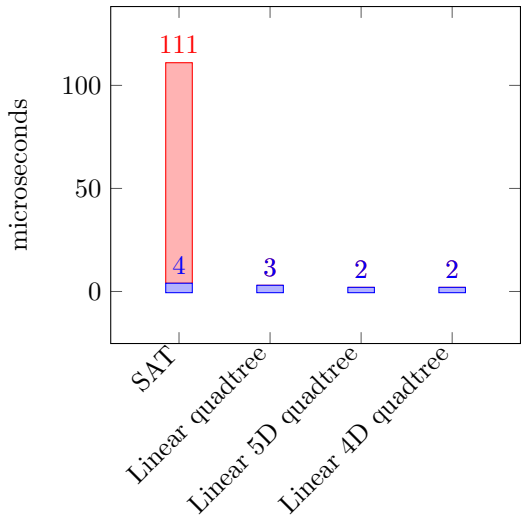


Figure 3.24: Comparision of the performance of SAT (including robot model in red) and the linear quadtree variants.

Binary classification

In this section we show the impact of the error of the discretization of the configuration space. While executing the linear quadtrees variants on the three test cases, we also executed the SAT algorithm which gave us the exact answer to the query. The SAT algorithm was invoked on robot models constructed with $D_a = 17.5$ and $H_a = 70$, since these values represent the physical robot the best. Recall from Section 3.2.7 that a false positive (FP) is a situation in which the linear quadtree reports that two robots are colliding, while there are not. A false negative (FN) is a situation in which the linear quadtree reports that two robots are not colliding, while they do collide. The number of false negatives must be zero, since it is obviously not allowed to let physical robots collide. The number of false positives should be as small as possible. Analogously, true positives (TP) are situations in which the linear quadtrees report that there is a collision, while the robots also collide. True negatives (TN) are situations in which the linear quadtree report no collision, and the robots indeed do not collide.

Further, we will introduce the notion of *accuracy*, which is:

$$\frac{TP + TN}{TP + TN + FP + FN}$$

k	**Samples**	**Collision samples**	D_a	H_a
32	33,663,204	584,416	30	82

Table 3.8: Parameters for the binary classification experiments.

To eliminate the false negative errors one can 'blow up' the robot model. The physical robots are best represented by $D_a = 17.5$ and $H_a = 70$, so we have included those test cases in Table 3.10,3.12 and 3.14. Since the linear quadtrees corresponding to these tables were constructed with $k = 32$, the error of the sampling procedure was then 16.5625 units. We blew up the robots models to $D_a = 30$ and $H_a = 82$ to eliminate the false negatives (we add at least $\sqrt{\frac{r^2}{2}}$ to D_a and H_a). The results for those test cases are presented in Table 3.9, 3.11 and 3.13. Two remarks can be made by comparing the linear quadtrees for these two different robot models. First, the accuracy of the linear quadtrees containing information from the blown up robots models is worse. This is because there is a higher amount of false positives. This is what we would expect: blowing up the robot model increases the chance of finding a collision while there is none. The second remark is that blowing up the robot model actually helps in test case T_3 (Table 3.14 and 3.13). In that case, also increasing the threshold t for the linear 4D quadtree seems to help: since more slices are grouped, more bits are added to the slices and these bits could, by coincidence avoid the false negative. In Figure 3.26 a visualization is given for the linear quadtree between two different robot models.

In all tables in this section, one can see that the classification information for the linear quadtree, linear 5D quadtree and the linear 4D quadtree for $t = 0$ is the same. This indicates, and this should be the case, that the amount of information contained in these variants is the same. For both the linear 5D quadtree and the linear 4D quadtree with $t = 0$ we reduced the memory requirements by different kinds of preprocessing, but we did not introduce errors.

One as to take care while interpreting the accuracy information. There are a few drawbacks of using this statistic. We choose to include it here to better show how quickly the accuracy can drop when t increases (for the linear 4D quadtree). One of the drawbacks is that it does not say anything about the amount of false positives versus false negatives. Having a few false positives is ok, but having a few false negatives is really bad. Further, having a decrease in the accuracy from 99% to 90% does not seem to be much, however one has to put this into context. The context is that a robot was stopped a few millimeters before the robots actually collide. Depending on the

actual application this might be ok or bad. The accuracy is also dependent on the test cases, as is clear from this section.

When t increases, one would expect the number of false positive to increase as well. This is because when t increases, a bigger error is allowed when matching slices, and the union of these 'less similar' slices becomes bigger. Therefore the chance of reporting a collision while there is none increases. Something strange seems to happen in, for example Table 3.10. At $t = 75$ the number of false positives was 548, while at $t = 100$ the number of false positives was 189. In this case the number of false positives thus decreases, something which is in contradiction with what we just stated. The explanation for this behavior is that the order of insertion of the slices, or actually the order of matching, is important. Consider slices $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$, as depicted in Figure 3.25. Here, the insertion order is first $\mathcal{S}_1$, then $\mathcal{S}_2$ and finally $\mathcal{S}_3$. Note that $|\triangle_{add}(\mathcal{S}_1, \mathcal{S}_2)| = 2$ and $|\triangle_{add}(\mathcal{S}_1, \mathcal{S}_3)| = 1$ and all $\triangle_{sub}$-differences are zero. When we set $t = 1$, $\mathcal{S}_1$ and $\mathcal{S}_2$ cannot be matched, their $\triangle_{add}$-difference is 2, hence bigger than one. Now $\mathcal{S}_1$ are placed in separate groups, G_1 and G_2. Next, $\mathcal{S}_3$ can be matched against $\mathcal{S}_1$ and will thus end up in group G_1.

Suppose that $t = 2$, then $\mathcal{S}_1$ and $\mathcal{S}_1$ can be matched against each other and they both end up in group G_1. Next, $\mathcal{S}_3$ cannot be matched against the union of $\mathcal{S}_1$ and $\mathcal{S}_1$, since $|\triangle_{add}(\mathcal{S}_1 \cup \mathcal{S}_1, \mathcal{S}_3)| = 2$, and G_{add} has been lowered to zero. The threshold G_{add} has been lowered from two to zero in the step in which $\mathcal{S}_1$ and $\mathcal{S}_2$ were matched, because their $\triangle_{add}$-difference was two (see Section 3.2.6 for the details about this algorithm).

Having explained why increasing t, might group slices differently, we can explain why increasing t might lead to a decrease in the number of false positives. This is because when querying a specific configuration, the corresponding slice $\mathcal{S}$ might be grouped with different slices. When $\mathcal{S}$ is grouped with different slices, different parts of the representative slice are added (or not). If the specific configuration needs to test exactly in the area of the slice which was not added (because of the different grouping), the query will return that there is no collision. When $\mathcal{S}$ was grouped such that the representative contained exactly these bits where the current configuration is about, the query will return that there is a collision.

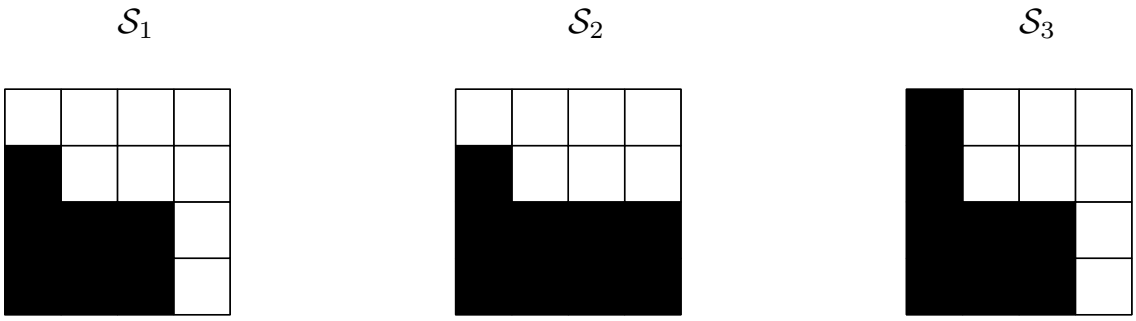


Figure 3.25: Three slices $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$.

Data Structure	TP	TN	FP	FN	Accuracy (%)
6D	821	4631	548	0	90.87
5D	821	4631	548	0	90.87
4D / t = 0	821	4631	548	0	90.87
4D / t = 5	821	4631	548	0	90.87
4D / t = 10	821	4520	659	0	89.02
4D / t = 25	821	4520	659	0	89.02
4D / t = 50	821	4223	956	0	84.07
4D / t = 75	821	4223	956	0	84.07
4D / t = 100	821	4587	592	0	90.13
4D / t = 125	821	3546	1633	0	72.78
4D / t = 150	821	3376	1803	0	69.95

Table 3.9: Accuracy information for testcase T_1 with $D_a = 30$ and $H_a = 82$ and $k = 32$.

Data Structure	TP	TN	FP	FN	Accuracy (%)
6D	821	5158	21	0	99.65
5D	821	5158	21	0	99.65
4D / t = 0	821	5158	21	0	99.65
4D / t = 5	821	5158	21	0	99.65
4D / t = 10	821	4990	189	0	96.85
4D / t = 25	821	4990	189	0	96.85
4D / t = 50	821	4990	189	0	96.85
4D / t = 75	821	4631	548	0	90.87
4D / t = 100	821	4990	189	0	96.85
4D / t = 125	821	5086	93	0	98.45
4D / t = 150	821	4520	659	0	89.02

Table 3.10: Accuracy information for testcase T_1, with $D_a = 17.5$ and $H_a = 70$ and $k = 32$.

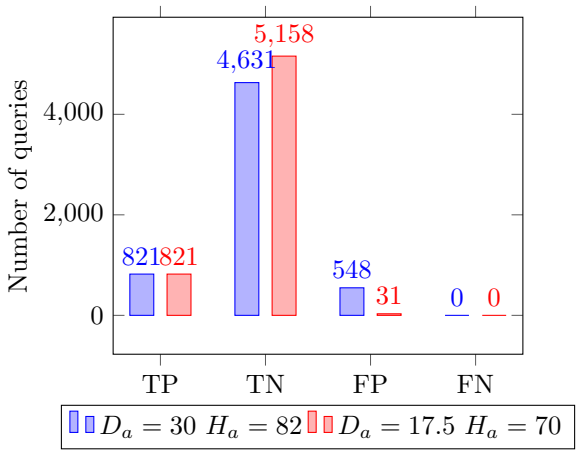


Figure 3.26: Binary classification for the linear quadtree with different settings for the robot model.

Data Structure	TP	TN	FP	FN	Accuracy (%)
6D	1485	4203	312	0	94.8
5D	1485	4203	312	0	94.8
4D / t = 0	1485	4203	312	0	94.8
4D / t = 5	1485	4203	312	0	94.8
4D / t = 10	1485	4203	312	0	94.8
4D / t = 25	1485	4203	312	0	94.8
4D / t = 50	1485	3108	1407	0	76.55
4D / t = 75	1485	3460	1055	0	82.42
4D / t = 100	1485	3108	1407	0	76.55
4D / t = 125	1485	3108	1407	0	76.55
4D / t = 150	1485	2743	1772	0	70.47

Table 3.11: Accuracy information for testcase T_2 with $D_a = 30$ and $H_a = 82$ and $k = 32$.

Data Structure	TP	TN	FP	FN	Accuracy (%)
6D	1485	4419	96	0	98.4
5D	1485	4419	96	0	98.4
4D / t = 0	1485	4419	96	0	98.4
4D / t = 5	1485	4419	96	0	98.4
4D / t = 10	1485	4419	96	0	98.4
4D / t = 25	1485	4419	96	0	98.4
4D / t = 50	1485	4054	461	0	92.32
4D / t = 75	1485	3445	1070	0	82.17
4D / t = 100	1485	3838	677	0	88.72
4D / t = 125	1485	3445	1070	0	82.17
4D / t = 150	1485	3445	1070	0	82.17

Table 3.12: Accuracy information for testcase T_2, with $D_a = 17.5$ and $H_a = 70$ and $k = 32$.

Data Structure	TP	TN	FP	FN	Accuracy (%)
6D	1581	4109	310	0	94.83
5D	1581	4109	310	0	94.83
4D / t = 0	1581	4109	310	0	94.83
4D / t = 5	1581	4109	310	0	94.83
4D / t = 10	1581	4109	310	0	94.83
4D / t = 25	1581	3744	675	0	88.75
4D / t = 50	1581	4019	400	0	93.33
4D / t = 75	1581	3353	1066	0	82.23
4D / t = 100	1581	3439	980	0	83.67
4D / t = 125	1581	3250	1169	0	80.52
4D / t = 150	1581	3830	589	0	90.18

Table 3.13: Accuracy information for testcase T_3 with $D_a = 30$ and $H_a = 82$ and $k = 32$.

Data Structure	TP	TN	FP	FN	Accuracy (%)
6D	1570	4179	240	11	95.82
5D	1570	4179	240	11	95.82
4D / t = 0	1570	4179	240	11	95.82
4D / t = 5	1581	4179	240	0	96
4D / t = 10	1581	4179	240	0	96
4D / t = 25	1581	4179	240	0	96
4D / t = 50	1581	4179	240	0	96
4D / t = 75	1581	4179	240	0	96
4D / t = 100	1581	4179	240	0	96
4D / t = 125	1581	4179	240	0	96
4D / t = 150	1581	3744	675	0	88.75

Table 3.14: Accuracy information for testcase T_3, with $D_a = 17.5$ and $H_a = 70$ and $k = 32$.

Conclusion

As described in detail in previous sections, the approach of using linear quadtrees to solve the collision detection problem is superior to the SAT approach regarding time performance: two or three microseconds versus 111 microseconds. One drawback of using linear quadtrees is their memory usage, since this is also limited in the NJ-controller. This is attacked by using two different approaches. These two approaches (linear 5D quadtree and linear 4D quadtree) can reduce the memory requirements by 45% and 90%, respectively, without any loss of information. When some loss of information is allowed (which is only applicable to the linear 4D quadtree), the reduction can be increased to 98%. The drawback of using the linear quadtrees is that they report false positives, i.e. the accuracy is not as high as using SAT (which is exact). For all test cases, the linear quadtrees for the robot model that was 'blown up' had at least an accuracy of 90%. While increasing the similarity threshold (for the linear 4D quadtree) this accuracy could drop quickly to around 70%. Whether this is actually a problem depends on the particular application. This is a consideration one has to make between speed, memory requirements and accuracy for a given application.

Chapter 4

Conclusions and future work

In this thesis we developed several algorithmic approaches to solve the collision detection problem for packaging robots. The class of packaging robots together with their use in packaging applications share some properties. Namely, the robots are mounted statically in their environment, are modeled by a few bounding volumes, and possible collision can only occur between a few robots. The collision detection problem for a setting in which these properties arise turns out not to be thoroughly studied. The available methods are used in mainly computer graphics and haptic technology. These methods are, however, not developed for use in a setting with real time collision detection for packaging robots. This application has more strict timing and memory requirements.

We showed how to model the complex structure of a packaging robot using geometric primitives. To make the process of computing this robot model more efficient, an improved and more efficient variant of the robot model was developed and implemented. Further, we implemented an existing and exact, but efficient algorithm (SAT) to perform collision detection. Its performance was measured in the real time system and it turns out that it can be used in the robotics software system as a primitive to perform collision detection.

To overcome the main part of the computational cost of this approach, we developed an algorithmic approach that exploits the use of a precomputed data structure. The data structure essentially contains already the answer to the collision detection problem at particular configurations. The configurations that this data structure contain are six-dimensional and therefore the memory requirements of this data structure can be high. To avoid this problem, we used a novel approach to ignore one or two dimension(s) from the six-dimensional configurations. The data structure then contains five- or four-dimensional configurations, respectively, and solves the local one- or two-dimensional problems during the query phase. It turns out that this approach can dramatically reduce the memory requirements, while not increasing the query time at all in practice (in some cases even an decrease in the query time was measured). Compared to the existing SAT algorithm, the precomputed data structure is superior regarding query time performance. Due to its superior performance, also this approach can directly be used in the robotics software system.

One can wonder how this novel technique of extracting dimensions from the data structure behaves with higher dimensional problems. Currently, we have three degrees of freedom for a single robot. The corresponding collision detection problem for a pair of robots is then six-dimensional. What if we do not have one pair of robots, but eight robots at the same time? Then we have 24-dimensional configurations. When having 24 dimensions, splitting the data structure in for example fourteen and ten dimensions might be another hard problem to solve. One interesting question is whether it is possible to build an hierarchy of reduced problems. In the case of 24 dimensions, one could split this in fourteen and ten dimensions. Further, the fourteen dimensions could be split in eight and six dimensions, which when again splitting are solvable (five and three and four and two dimensions). The ten-dimensional problem can then again be split into a six-

and four-dimensional problem. More interesting questions pop up when one starts to wonder the amount of dimensions that should be ignored from the data structure. For example, is it better to split the 24-dimensional problem into a fourteen- and ten-dimensional problems, or should one consider the 18- and six-dimensional problem? Another question is what dimensions should be extracted. As shown in the experimental evaluation, choosing different dimensions to ignore from the data structure results in a different reduction of the memory requirements.

The answers to these questions are unclear at this moment, but it would be important to look at the properties of the configuration space. Exploiting monotonicity can help in deciding which dimensions to ignore. Further, using different techniques to ignore the dimensions can be very important. We presented a technique based on similarities, but many variants are possible here. Basically a large part of (existing) clustering techniques can be used. Further, one can think of interpreting the manifolds in the ignored dimensions. If there is some structure, some (simple) approximated geometric primitives can be used to describe the local problem and thereby potentially reducing the amount of memory needed to store this problem.

If this novel approach is also applicable to higher dimensional problems, efficient collision detection can be performed for robots with more degrees of freedom or even for more than two robots at a time.

Another, more practical, application is using this approach in collision avoidance. In collision avoidance, one generates (local) trajectories, such that the robots do not collide, but they try to avoid each other. A first primitive in this system is having a procedure to solve the collision detection problem. The presented approaches to solve the collision detection problem can directly be used here. Some of the existing techniques try to generate some roadmap that captures the connectivity of the configuration space. Another, more interesting question is whether the information that the precomputed data structure contains, can be transformed into such a roadmap. One of the techniques that could be used to do this is that of computing the dual of a quadtree. The dual of a quadtree is a graph that basically already represents the connectivity of the configuration space. The nice thing about using the dual of a quadtree is that this can also be precomputed and could potentially also lead to very efficient generation of (local) paths during system operation.

Bibliography

[1] R. Clavel. *Conception d'un robot parallle rapide 4 degrs de libert.* PhD thesis, Lausanne, 1991. 31

[2] J.J. Craig. *Introduction to Robotics: Mechanics and Control.* Addison-Wesley series in electrical and computer engineering: control engineering. Pearson Education, Incorporated, 2005. 16, 18

[3] M. de Berg. *Computational Geometry: Algorithms and Applications.* Springer, 1997. 36

[4] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, December 1982. 36, 38

[5] E.G. Gilbert, D.W. Johnson, and S.S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *Robotics and Automation, IEEE Journal of*, 4(2):193–203, Apr 1988. 8

[6] S. Gottschalk, M. C. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 171–180, New York, NY, USA, 1996. ACM. 8, 25

[7] S. Har-peled. *Geometric Approximation Algorithms.* American Mathematical Society, Boston, MA, USA, 2011. 36

[8] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. Octomap: an efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, 2013. 8

[9] J. Huynh. Separating Axis Theorem for Oriented Bounding Boxes. `www.jkh.me/files/tutorials/SeparatingAxisTheoremforOrientedBoundingBoxes.pdf`, 2008. [Online; accessed 6-August-2014]. 27

[10] P. Jimnez, F. Thomas, and C. Torras. 3d collision detection: A survey. *Computers and Graphics*, 25:269–285, 2000. 6

[11] D. Jung and K.K. Gupta. Octree-based hierarchical distance maps for collision detection. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 1, pages 454–459 vol.1, Apr 1996. 8

[12] S.M. LaValle. *Planning Algorithms.* Cambridge University Press, 2006. 16

[13] M. Lin and J. F. Canny. A Fast Algorithm for Incremental Distance Computation. In *International Conference on Robotics and Automation*, 1994. 8

[14] M. C. Lin and D. Manocha. Collision and proximity queries, 2003. 6

[15] B. Mirtich. V-clip: Fast and robust polyhedral collision detection. *ACM Trans. Graph.*, 17(3):177–208, July 1998. 8

[16] G. van den Bergen. *Collision Detection in Interactive 3D Computer Animation.* PhD thesis, Eindhoven University of Technology, 1999. 6

[17] G. Van den Bergen. A fast and robust gjk implementation for collision detection of convex objects. *J. Graph. Tools*, 4(2):7–25, March 1999. 8, 25

[18] R. Weller. *New geometric data structures for collision detection.* PhD thesis, Dissertation, Universität Bremen., 2012. 6

Appendix A

Project execution

In this chapter we will shortly describe the organization of the project. The project has been performed at Omron in Barcelona, Spain. At this office of Omron the teams work using an Agile software development methodology. An Agile software development methodology focusses on quickly developing software, while being able to anticipate on changing requirements. The specific kind of methodology that was adopted is the SCRUM methodology. Because of this the project had also to be executed according to the SCRUM rules.

The project was executed in *sprints* of two weeks. A sprint is a small period in which a part of the project is executed. Before the start of a sprint, it is planned what tasks will be executed, along with a size estimation and priority. During a sprint there are daily standup meetings. These meetings take usually about 15-20 minutes and during these meetings one tells what he did the day before and what he plans to do today. Also if some problems arise, one can share this with the other team members. Apart from these standup meetings there are some meetings at the last day of the sprint. On that day, there is a 'done' show, in which people show what they accomplished during the sprint (if the results are interesting/suitable for visualization). Further, there is a retrospective in which everybody reflects the sprint: what went well and what did not? After a quick summarization of every team member, the topics are grouped and everybody votes for a particular group. The most popular group is then discussed and possible improvements are investigated.

Next to these meetings, it turned out to be helpful to have additional meetings with team members to synchronize the work and to avoid possible problems. On Thursday of the first week of the sprint and on the Tuesday of the second week of the sprint we kept these additional meetings. In these meetings we discussed what has been done (in more detail than at the daily standup), what problems arise and what will be done until the next meeting.

Next to these internal meetings, at every Tuesday we organized meetings with the graduation supervisors, Kevin Buchin and Rodrigo Silveira. These meetings are intended for a detailed update about the status of the project and to discuss approaches on a theoretical level.

A.1 Implementations in the NJ-controller

In this section we will describe how both the SAT algorithm, together with the implementation of the robot model was implemented in the robotics system. Further, we implemented the linear quadtree and the linear 5D quadtree into the robotics system. In FigureA.1, the different layers in the robotics system are depicted. The rectangle called 'Collision project', is a (ANSI) C project that include the code for the SAT algorithm, together with with implementation of the robot model. It is contained in a Program Organizational Unit (POU) that includes logic such as initialization and destruction during system startup and termination. Such a POU is then contained in a Function Block, which is another layer of abstraction used in the robotics system. This layer contains more high-level logic, such as a state machine to control the execution of the POU inside

the robotics system.

Since the code in the collision project needs access to the direct and inverse kinematics engine, as well as some other information, a robotics API was constructed. This API provides an easy interface to obtain this information.

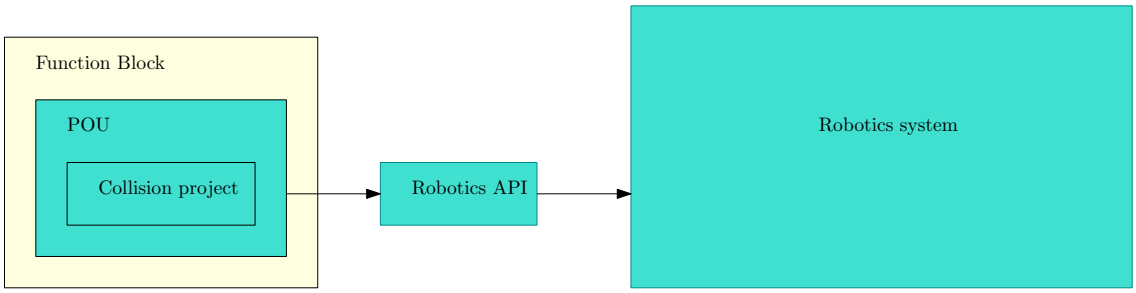


Figure A.1: An overview of the project implementation inside the robotics system.

Since deploying the code inside the robotics system is a tricky task, multiple levels of testing are used. First of all, the code itself is tested at a low level using unit tests. Next, parts of the robotics system are 'mocked': implemented such that they mimic the behavior of the real implementation. The reason to use these mocked implementations is that one is able to capture some output when a piece of code fails to do its job. If one would omit this step and directly deploy the code inside the robotics system, it is extremely hard to debug the code when an error occurs. This because it is impossible to use a debugger in this stage, and capturing output in this stage is unreliable—the robotics system might crash if this takes too long. When the code seems to be correct in the mocked environment, one can deploy the code in the real robotics system and start testing.

Before starting to test the deployed code while using physical robots, another simulator is used. The code is executed inside the robotics system on the NJ-controller, but instead of providing the outputs to hardware that is able to control the robots, the output is directed to this simulator. The simulator is part of the IDE SysmacStudio that is used to implement robotics applications.[1]

While developing the robot model, it is useful to have some visualization of this model. While developing the robot model directly in (ANSI) C code a visualization is hard to establish (one has to write a simulator). Therefore the freeware software packet Scilab is used.[2] The SAT algorithm and the procedures to construct the robot model were first implemented in this environment before implementing them in (ANSI) C.

[1] http://industrial.omron.eu/en/products/catalogue/motion_and_drives/machine_automation_controllers/software/sysmac_studio/default.html

[2] http://www.scilab.org/