Dynamic real-time collision detection for packaging robots using monotonicity of configuration subspaces

Ronald van Zon[1], Diego Escudero[2], Dan Halperin[3], Igor Jovanovic[2], Raffaelle Vito[2], Rodrigo I. Silveira[4*] and Kevin Buchin[1*]

***Abstract*— Due to the increasing demand for higher performance (throughput and efficiency) of robotic packaging systems, and the need to keep the system's footprint as small as possible, robots must operate closer to each other. This gives rise to progressively more difficult variants of 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 identify certain monotonicity properties in the composite configuration space of the underlying robots, which help us to devise highly effective collision detection algorithms for them. We lay out the theoretical basis of our development, describe the new algorithms, and report on simulation results. The results show that two of the new algorithms can reduce the memory requirements by 45% and 90%, respectively, without any loss of information as opposed to an approach based on solely existing techniques. Furthermore, our solution easily satisfies the real-time constraint with a typical query time of 3 microseconds. Finally, we demonstrate the real-time capabilities of our algorithms in an industrial setup.**

I. 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 makes extensive use of robots for these applications (see Figure 1).

The packaging industry is always looking for new systems that allow them to increase the amounts of objects that can be packed during a fixed time frame. One way to accomplish this is by placing more robots along the conveyor belts. One of the problems with this approach is that it usually means that the robots are placed more closely to each other (the length of the conveyor belt can in general not be extended too much). This gives rise to the problem that robots could collide with 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 fundamental primitive in this system is to be able to detect when robots are about to collide. This problem is called *collision detection* and will be the topic of this paper. The system in which this primitive will be used is built on top of a real-time operating system, which is common in modern controllers for robotics applications.

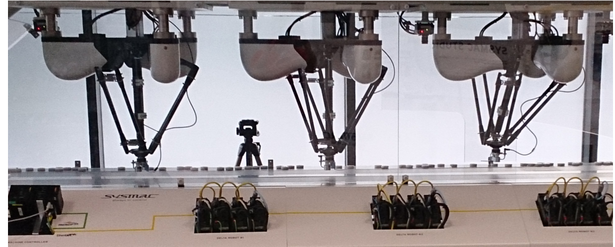

Fig. 1: Three Delta3-R robots in a test setup.

A. Setting and system properties

The applications in the packaging industry vary widely, e.g., packing (soft) food is different from packing metal screws. Furthermore, the components of the packaging system may vary between factories even for the same 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 2 we have depicted a schematic version of a typical setup. Here, C_1 and C_2 denote the two conveyor belts and R_1 and R_2 denote the mounting positions of the two 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 and R_2) are statically mounted.
- 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

*Equal last contributor

[1]Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands ronaldvanzon1@gmail.com, k.a.buchin@tue.nl

[4]Departamento de Matemática, Universidade de Aveiro, Aveiro, Portugal, Departament de Matemàtica Aplicada II, Universitat Politècnica de Catalunya, Barcelona, Spain rodrigo.silveira@upc.edu

[3]School of Computer Science, Tel Aviv University, Tel Aviv, Israel danha@tau.ac.il

[2]Omron, Barcelona, Spain {diego.escudero, igor.jovanovic, raffaele.vito}@eu.omron.com

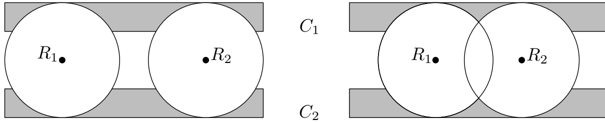

Fig. 2: Current (left) and desired (right) application

moving, or stopped while packing the objects into the container.
- The size of the system, that consists of the conveyor belts, robots and other parts is kept 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 for Delta3-R robots). Although the theory in this paper has been developed such that it is applicable to multiple (different) robot types, we will focus on the Delta3-R type. The reason being that they are considered to be the fastest robots in their field, and thus would have the most restrictive real-time constraints.

The problem studied in this paper can be stated as follows: *Solve the collision detection problem making use of the following properties and under the listed constraints.*

Both the Scara and Delta robots share some properties in the context of being used in the packaging industry:

- The robots are statically positioned in the environment and typically their base cannot move.
- The robots are placed such that potential collisions can only occur between a very small number of robots (typically two or three).
- The robots can be faithfully represented by a few simple bounding volumes.

A solution for the collision detection problem has to be implemented in a robotics software system, which runs on an embedded system. The constraints imposed by this system are the following:

- Collision detection has to be performed every millisecond and within ~40 microseconds.
- Memory requirements must be at most tens or a few hundreds of megabytes.
- Only the current and next position of robots are known.

B. Related work

Current techniques for collision detection, like the GJK algorithm [3], [7] are not suitable since they are designed for uses that don't impose such stringent time constraints as, e.g., in computer graphics for which a typical time constraint is 30 milliseconds. For an extensive overview of collision detection, we refer to [6]. This work is based on a M.Sc. thesis [8] and is supported by a website[1] containing supplementary material. For lack of space we omit many technical details, and in particular the detailed description of several algorithms—the interested reader could find all the omitted material in the thesis [8].

[1]http://www-ma2.upc.edu/rsilveira/collisiondetection

II. Geometric modeling and robot model

Since the Delta3-R robots have a somewhat complex structure—they consist of springs, motors, rectangular arms, bolts, etc.—it pays off to develop an approximate geometric model of the robot. This model will represent the important parts of the robot (in the context of collision detection) and disregard other parts. To model the robot we use *bounding volumes*. In particular, we use *Oriented-Bounding-Boxes* (OBBs) to model the arms of a Delta3-R robot. Since only the lower parts of the arms of the Delta3-R robots can collide with each other (due to the mounting, the upper parts of the arms cannot collide), only these parts are modeled with OBBs in the robot model. The robot model thus consists of three OBBs, one for each (lower part of each) arm.

III. Separating Axis Test

To compute whether two robot models collide, we have to check for each pair of OBBs (one OBB from each robot model) whether they collide. We choose the *Separating Axis Test* (SAT) variant from Gottschalk et al. [4] which is described detailed in [5]. The reason for choosing this algorithm is that the technique is numerically robust even in the presence of degeneracies. Furthermore, it is possible to compute within 200 arithmetic operations whether a collision takes place [4]. Finally, the algorithm is simple to implement.

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 and only if A and B do not intersect, it is possible to define such a separating plane between them. 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 (on which the projections of A and B are disjoint), we also found a separating plane. For a pair of OBBs there are only 15 possible separating axes [4].

The basic variant of the SAT algorithm can be optimized in several ways, as described in [5], but can also be optimized by caching the separating axis. In this paper, however, we do not implement this optimization.

IV. Spatial (hierarchical) decomposition

A drawback of the method used in Section III is that for every problem instance, one needs to construct two robot models and invoke the SAT algorithm. It turns out that, compared to the extremely fast SAT algorithm, the construction of the robot model is a relatively slow operation (see Section V). To avoid constructing the robot model for every query we need 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 precomputed robot models and store the answers to the queries in the data structure. Then, during system operation, one has to query the data structure to retrieve the answer to the collision detection problem. Another advantage of this approach is that it exploits the multi query problem arising here: we can re-use this data structure for every query during system operation.

A. Discretization of the configuration space

Ideally, one queries the data structure during system operation to check whether the robots at their current positions are in collision. The current positions of the robots are denoted by a configuration $c = (x, y, z, x', y', z)'$, where the positions (x, y, z) and $(x', y', z)'$ denote the positions of the reference points of both robots within physical space. The data structure thus has to be indexable by configurations. 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.

Since $\mathcal{CS}$ is a continuous space we have to discretize it. 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. Further, this approach has the advantage of storing samples from 'every' part of $\mathcal{CS}$, so there are no parts of $\mathcal{CS}$ that are not covered.

For every sampled configuration, we construct the corresponding robot models and invoke the SAT algorithm on them. Along with the sample, we store whether it represents a collision or non-collision configuration.

B. Linear 6D quadtree

A quadtree is a rooted tree in which the 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 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. The quadrants represented by the leaves 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).

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 quadrant, since there is no further subdivision there. In our setting a quadtree adapts naturally to the grid structure of the discretization of $\mathcal{CS}$. Further, we expect collision samples to be grouped, which allows the quadtree to store this information compressed, using *condensation* [2].

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). A node is further subdivided into subnodes as long as its hypercube contains more than one sample. For a more detailed introduction to quadtrees and its properties, see e.g., [1].

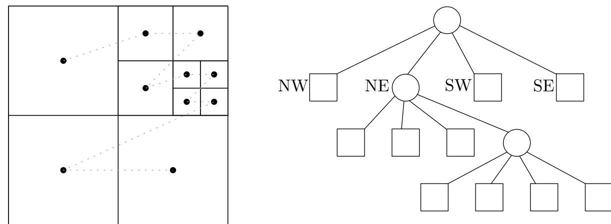

Fig. 3: A quadtree and the corresponding subdivision of a square. Dotted line is the z-order curve that determines the order for generating indices for the squares of the quadtree.

A drawback of using quadtrees is the space overhead caused by storing interior nodes. A more memory efficient version of a quadtree, a *linear quadtree* exists that indexes the leaves (containing labeled configurations in our case) by *location codes*. It thus omits storing the interior nodes. An introduction to linear quadtrees can be found in [2].

Next we present the main parts of our method: constructing the quadtree and querying it.

1) Constructing the linear 6D quadtree: The algorithm to construct the linear quadtree is depicted in Algorithm 1. Since we only store collision configurations, the algorithm is given a set of samples, where every sample represents a collision configuration. For each sample, a location code is generated (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 (line 6).

Algorithm 1 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 height h of the linear quadtree must be known prior to constructing the linear quadtree because it is needed to compute the location codes to align the discretization resolution with the subdivision that the linear quadtree induces.

We want to place one 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. This will only be correct if we have sampled at least once in every hypercube that the subdivision of the linear quadtreee induces.

2) Querying the linear 6D quadtree: During system operation, we want to determine whether two robots at a particular configuration collide. The procedure is a binary search on the (sorted) location codes. One detail is that the location codes can be condensed and an 'ordinary' binary search might fail. Some extra checks that do not increase the asymptotic time complexity of the query can be added to solve this.

C. Linear 5D quadtree

To further reduce the memory requirements, we investigate the idea to 'split off' some dimensions in the linear quadtree (and solving the problem in these 'split off' dimensions later). More specifically, if one dimension is split off 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 the dimension that was split off. 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 to fixing the pose of one robot by fixing the coordinates (x, y, z) and fixing the pose of the second robot 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. If the collection of samples would have some structure, we could save space in storing these z'-coordinates.

A central assumption we make that will support this approach is the following: *The composite configuration space of two robots is mostly monotonic*. Monotonicity here means that collision configurations are grouped consecutively in a dimension. It is thus improbable to have multiple alternations between collision and non collision configurations along one dimension, when all other five dimensions are fixed. A justification (apart from experimental verification) is the 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. In these situations, the corresponding configurations that represent collisions will be grouped consecutively in $\mathcal{CS}$.

To exploit this monotonicity, we take the following approach. We construct a linear 5D quadtree on five dimensions of every configuration. Now, at each leaf ν we have a collection of configurations that only differ on the dimension that was split off. Let the *collision interval* be the (sub)interval that contains a maximal consecutive group of collision configurations in this dimension. At leaf ν, we now store the begin- and end-coordinate of this collision interval, instead of all collision configurations separately. Since there might be more than one group (as $\mathcal{CS}$ is not perfectly monotonic), we store the corresponding coordinates of all these groups if needed.

The query procedure will be the same as for the linear 6D quadtree, with the difference that when finding a particular location code, we have to check whether the current value of the dimension that was split off is contained in (one of) the collision interval(s) stored at this location code.

D. Linear 4D quadtree

The linear 4D quadtree takes the approach from the linear 5D quadtree one step further. In the linear 4D quadtree we split off 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 split off dimension. A 2D slice is thus a grid on which the cells can be black: the configuration results in a collision, or white: the configuration does not result in a collision.

We again make the assumption that $\mathcal{CS}$ is mostly monotonic, which allows us to compress the 2D slices. Monotonicity in this 2D case will mean that the regions in the 2D slices will be *orthogonally convex*. That is, a segment parallel to any of the two coordinate axes connecting two points in such a region will be entirely inside this region. The reason it helps (implicitly) is that the number of different (ortho-convex) slices that are possible under the monotonicity constraint is drastically lower, see Table I. There, *strong monotonicity* means that the collision configurations start at the beginning of the intervals (1D) or at a corner of a 2D slice. Therefore there are less possible collision intervals and slices. Further, the differences between two slices cannot be in the entire space of the slice, but would rather be close to the boundary (otherwise the monotonicity property would not hold).

Dim.	Strong monotonicity	Monotonicity	None
1D	m	$\binom{m}{2} + m + 1$	2^m
2D	$\binom{2m}{m} \approx \frac{4^m}{\sqrt{\pi m}}$	$\binom{2m}{m}^4 \approx \left(\frac{4^m}{\sqrt{\pi m}}\right)^4$	2^{m^2}

TABLE I: Upper bounds on the number of different (ortho-convex) intervals (1D) and slices (2D) of m cells in every dimension that are possible under the imposed constraints.

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 paper, 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.

1) 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 S as being a rectangular arrangement of cells, in which the cells represent collision configurations. The threshold t is related to the cardinality of this difference $|A \triangle B|$.

2) Constructing and querying the linear 4D quadtree: In order to ensure a somewhat compact representation and constant complexity access time for the cells of a slice we use *bit arrays*. A bit array is a list of integers, the bits of which can be addressed independently. Each bit thus represents a cell.

We construct a linear 4D quadtree on four dimensions of every configuration. Now, at each leaf ν we have slices of configurations that only differ on the two dimensions that were split off. We store each slice into a central *slices storage* if it differs more than threshold t from all currently stored slices (using the similarity metric discussed above). A pointer to this slice is then stored at the location codes.

The query procedure will be the same as for the linear 6D quadtree, with some differences to 'solve' the additional two-dimensional problem. At the obtained location code, we retrieve the corresponding slice using the stored pointer. Then, we check the bit in the slice that corresponds with the two values of the dimensions that were split off. If it represents an collision we can immediately return that there is a collision, and vice versa.

3) Memory requirements: In our implementation, the linear quadtree variants are implemented as C structs. To analyze the amount of memory that is used by the linear quadtree, we make the following assumptions:

1) A `double` resides in eight bytes.
2) A `int` resides in four bytes.
3) A `unsigned long long int` resides in eight bytes.

The memory requirements of the linear variants are depicted in Table II.

E. 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 focus on one of the

Data structure	Memory requirement (bytes)
Linear 6D quadtree	$8m + 64$
Linear 5D quadtree	$12m + 16i + 64$
Linear 4D quadtree	$8m + \frac{sb^2}{8} + 8b + 76$

TABLE II: The memory requirements of the three variants. Here, m represents the amount of location codes, i the total amount of intervals, s the total amount of slices and b^2 the amount of cells contained in a single slice.

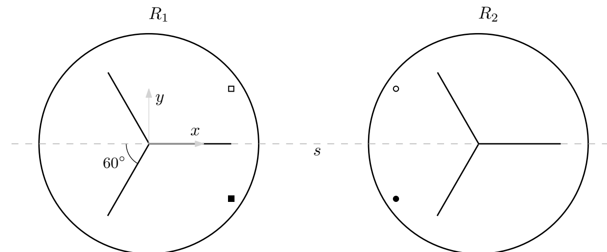

Fig. 4: A xy-projection of two Delta3-R robots.

most common placements for Delta3-R robots, depicted in Figure 4. In this setup, two robots, R_1 and R_2 are depicted.

The symmetry that can be exploited here is a reflection in axis s. When the reference point p_1 of R_1 is placed at the box and the reference point p_2 of R_2 is placed at the small circle, they will collide if and only if they collide also when p_1 is placed at the square and p_2 is placed at the disk. More formally, a configuration $c = (\Box_x, \Box_y, z, \bigcirc_x, \bigcirc_y, z')$ is a collision configuration if and only if the configuration $c' = (\Box_x, -\Box_y, z, \bigcirc_x, -\bigcirc_y, z')$ is a collision configuration too (the subscript on the symbols denote the coordinate in the corresponding dimension). The configuration c' is depicted in the figure as $c' = (\blacksquare_x, \blacksquare_y, z, \bullet_x, \bullet_y, z')$. Thus 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.

V. Experiments

The proposed methods were implemented in a setup that contains two Delta3-R robots and experiments were run on the controller. We present three perspectives: memory requirements, query times and accuracy.

The code to generate the linear quadtrees was written in C++ and ran on a laptop with Windows 7 64 bits SP1 on a Core i7 2860QM 2.50GHz processor with 8GB of memory. The code to query the linear quadtrees was written in C, compiled with GNU GCC 4.3.3 with flags `-O2, -w1`.

1) Memory requirements: The linear 5D quadtree was constructed such that at the leaves the z' dimension is stored. The linear 4D quadtree was constructed on zz'-slices, where each slice contains $11^2 = 121$ configurations.

In Table III, we show the results of the generated linear quadtrees. As can be seen, splitting off dimensions reduces the memory requirements since much less location codes have to be stored. The extra memory required to store the

split off dimensions is less than the memory gained by the fewer location codes. Increasing threshold t (that limits the amount of cells that do not match between two slices) for constructing the linear 4D quadtree also helps reducing the memory requirements. For $t = 0$, exact matches are required between slices (so there is no information loss) and the memory requirements are already reduced by ~90%.

Data Structure	#Location codes	Memory (bytes)	#Slices
6D	466,984	3,735,936	-
5D	85,539	2,053,100	-
4D / t = 0	8,548	380,884	2,011
4D / t = 5	8,548	242,244	278
4D / t = 10	8,548	229,444	118
4D / t = 25	8,548	222,164	27
4D / t = 50	8,548	220,644	8
4D / t = 100	8,548	220,164	2
4D / t = 150	8,548	220,084	1

TABLE III: Results for the linear (6D, 5D, 4D) quadtree with zz'-slices. Memory use was computed based on Table II.

In Figure 5, we show the memory use of the linear 4D quadtree for the same situation, as well as a linear 4D quadtree generated on yy'-slices. The behaviors are the same, although there was a better memory reduction achieved with yy'-slices. More importantly, there is a drastic difference in memory use for $t = 0$: the approach using zz'-slices uses ~30% of the space required by the approach using yy'-slices.

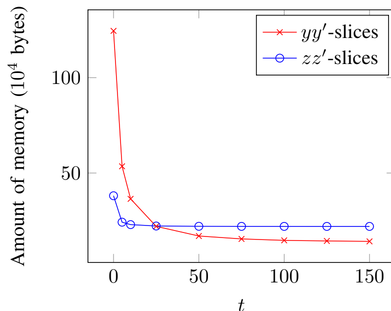

Fig. 5: The amount of memory (10^4 bytes) of the linear 4D quadtree when using yy'- or zz'-slices.

2) Time performance: The time to execute the SAT algorithm, including constructing the two robot models, is 111 microseconds (μs) of which SAT takes four. The linear quadtree variants had query times of 3 μs, 2 μs and 2 μs for the 6D, 5D and 4D variants, respectively. Therefore we can conclude that the linear quadtree approach is much faster. Also the extra computational cost for solving the one- and two-dimensional problems in the 5D and 4D variants is neglectable when compared to the 6D variant. This is because there are less location codes to perform a binary search on.

3) Accuracy: The linear quadtree approaches only approximate $\mathcal{CS}$ and thus can return some wrong answers. Since the SAT algorithm will always give an exact answer, we compared the answers of these two approaches and classified them: *false negatives* (FNs) are answers that there is no collision while in fact there is and *false positives* (FPs) are answers that there is a collision while there is not. At all cost FNs should be eliminated, while FPs should be minimized. To eliminate FNs we increased the dimensions of the OBBs of the robot model according to the resolution of the quadtree. Our experiments verify that the number of FNs is zero, while there are 21 FPs for all three variants (corresponding to answering 99.65% of the queries correctly). When increasing threshold t, the number of FPs increases up to 659, corresponding to answering 89.02% of the queries correctly. More extensive experiments are reported in the thesis [8].

VI. Key directions for further research

The approaches described in this paper for *collision detection* perform very well. The constraints on the query time and memory requirements are met easily, while accuracy can be kept high. Since queries can be performed extremely fast it is definitely worth investigating how to do (local) *motion planning* with these approaches. Moreover, since the memory requirements can be made very small it is really interesting to investigate how to tackle higher dimensional problems with this approach (e.g., a 24-dimensional configuration space).

Acknowledgment

We would like to thank Oren Salzman and Kiril Solovey for their ideas in this work and specifically Oren for his ideas about exploiting symmetries.

R. S. was partially supported by projects MINECO MTM2012-30951/FEDER, Gen. Cat. DGR2014SGR46, and by Portuguese funds through CIDMA and FCT, within project PEst-OE/MAT/UI4106/2014, and by FCT grant SFRH/BPD/88455/2012.

K. B. was partially supported by the Netherlands Organisation for Scientific Research (NWO) under project no. 612.001.207

Work by D. H. was partially supported by the Israel Science Foundation (grant no. 1102/11), by the German-Israeli Foundation (grant no. 1150-82.6/2011), and by the Hermann Minkowski–Minerva Center for Geometry at Tel Aviv University.

References

[1] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

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

[3] 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.

[4] 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.

[5] J. Huynh. Separating Axis Theorem for Oriented Bounding Boxes. `www.jkh.me/files/tutorials/SeparatingAxisTheoremforOrientedBoundingBoxes.pdf`, 2008. [Online; accessed 30-September-2014].

[6] M. C. Lin and D. Manocha. Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*. Chapman and Hall/CRC, 2nd ed. edition, 2004.

[7] 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] R. van Zon. Dynamic real-time collision detection for packaging robots. Master's thesis, Eindhoven University of Technology, 2014. `http://www-ma2.upc.edu/rsilveira/collisiondetection/paper.pdf`.