UNIVERSITÀ DEGLI STUDI DI PADOVA

—

DEPARTMENT OF INFORMATION ENGINEERING
DEPARTMENT OF MANAGEMENT AND ENGINEERING

—

MASTER'S DEGREE IN
AUTOMATION ENGINEERING

INDUSTRIAL MANIPULATORS COLLISION DETECTION ALGORITHMS

SUPERVISOR: CH.MO DR. GIULIO ROSATI

STUDENT: MARCO FERRARO

Academic Year 2014/2015

Marco Ferraro: *Industrial manipulators collision detection algorithms,* Analisi delle collisioni tra manipolatori industriali: algoritmi di collision detection, © March 2015

LOCATION:
Padova

TIME FRAME:
September 2014 - March 2015

Dedicated to my family.

Abstract

In this work we present some algorithms for detecting any collision between two robot manipulators. Collisions can be detected only if the trajectory of both manipulators is known, so on the first part we develop a procedure to estimate robot trajectories given only via points, robot and workcell configuration. This phase is carried out considering an Epson® C4A601S anthropomorphic manipulator, but it may be adapted to other robots as well. On the second part, we develop the actual general algorithm to detect collisions, providing multiple models of each robot link which differ in reliability and simplicity. A user interface is developed to produce one of the models which is employed in this phase. The algorithm is then optimized for anthropomorphic robots, in order to be performed on-line. Finally, some results on critical situations are summarized, which show the effective behaviour even in worst case conditions.

Acknowledgments

I would like to thank all the people who contributed to the realization of this work.

First of all I want to thank my supervisor, dr. Giulio Rosati, who gave me the possibility to develop and carry out this work, which I consider as a proud milestone in my life. I also want to thank Pd.D student Luca Barbazza, who helped me during many phases of this work, and the Department of Management and Engineering for giving me the possibility to carry out the experimentations in its laboratories.

Special thanks to all the people that supported me during this thesis, my university mates Matteo, Gianluca and Riccardo, with whom I shared all these last years, and all my friends who helped me to get to this point, with whom I share the best moments in my life from years now.

My best gratitude finally goes to my relatives and my family, my brother Stefano, who has always been an incentive to do my best, and mostly to my parents, Dennis and Raffaella, who allowed me to pursue this career and supported me for every day of my life. If I got to this milestone, it's all your merit.

Contents

1 INTRODUCTION 1
- 1.1 Historical background 1
- 1.2 Environmental background 2

2 TRAJECTORY ESTIMATION 5
- 2.1 Problem statement 5
  - 2.1.1 The `Go` instruction 6
  - 2.1.2 The `Jump3` instruction 8
  - 2.1.3 The Continuous Path (CP) command 9
- 2.2 Approximation of a `Jump3` motion with `Arch` option 10
  - 2.2.1 Estimation of continuous trajectory 10
  - 2.2.2 Approximation using knots smoothing 12
- 2.3 Approximation of the junction between two PTP motions 17
  - 2.3.1 Path sampling according to acceleration and deceleration times 18
  - 2.3.2 Path sampling without robot querying 19
  - 2.3.3 Other possibilities and improvements 21
- 2.4 Experimental results on trajectory estimation 22

3 COLLISION DETECTION 31
- 3.1 Solid interpenetration using Swept Sphere Volumes (SSV) 32
  - 3.1.1 Interpenetration between two linear SSVs (LL algorithm) 34
  - 3.1.2 Interpenetration between linear and polygonal SSV (LP algorithm) 35
  - 3.1.3 Interpenetration between two polygonal SSVs (PP algorithm) 40
  - 3.1.4 Complexity of SSV modelling algorithm 41
- 3.2 Solid interpenetration using boxes 43
  - 3.2.1 Complexity of collision detection algorithm using boxes 50
- 3.3 User interface for initial SSV setup 51
  - 3.3.1 SSV automatic generation procedure 55
- 3.4 Collision detection general algorithm 57
  - 3.4.1 Collision between two static joint configurations 58
  - 3.4.2 Collision detection between trajectories 62

4 ALGORITHM OPTIMIZATION 65
4.1 Robot Maximum Swept Volume approach 66
4.1.1 Box obstruction according to first joint position 66
4.1.2 Swept Volume computation 68
4.1.3 Collision detection between robot swept volumes 72
4.2 Group of Joints approach 76
4.3 Algorithm outline 78
4.3.1 Trajectory subdivision 78
4.3.2 Combination between different abstraction levels 79
4.3.3 High-level collision-free algorithm 80
5 EXPERIMENTAL RESULTS 83
5.1 Reliability and precision of the algorithm 83
5.2 Implementation details 84
5.3 Modelling Set-up 84
5.3.1 SSV modelling 84
5.3.2 Box modelling 85
5.4 Critical situation test 86
5.5 Simulation results 88
6 CONCLUSIONS 93
A TRAPEZOIDAL-SPEED LAW RELATIONSHIPS 95
B PROPERTIES OF TRANSFORMATION MATRICES 97
C KINEMATIC MODEL OF ROBOT MANIPULATORS 99
BIBLIOGRAPHY 103

1

Introduction

Industrial robots have made a significant improvement in manufacturing processes automation over the last decades. Low production costs and productivity increase are two main reasons why they have been used, so that it is common to find industrial environments where more robots are employed for the same production line. This can be done in an efficient way by sharing the same workspace with more robots, in order to reduce the size of the workcell and eventually to cooperate between different manipulators.

This work aims to build a system which is capable of detecting any collision between two robot manipulators which share the same workspace. In particular, we want to perform this on-line, in order to not alter workcell performance. The system we are going to build does not use any external agent to determine collisions, such as cameras or sensors, but it only needs to know the trajectory each robot is going to follow.

Since our algorithm will be performed on-line, this system can fit in a flexible robotics manner, as the work cycle can be always different.

1.1 Historical background

Many works which matters with collision detection and trajectory collision-free planning have been developed over the years. Firstly developed system had to cope with lack of computational resources, whereas over time the technological progress gave the possibility to overcome these problems and open the road to increasingly sophisticated systems which could cope with high-dimensional problem with many degrees of freedom.

Starting from 1987 Lee *et al.* [9] developed a solution which exploits a *collision map* between only last 3 links of a 6-axes manipulator, and solved the planning problem with a *time scheduling* technique, which involves speed reduction and delay time superimposing. A similar solution was proposed in 1989 by O'Donnell *et al.* [11], where a similar approach to the collision map called *Task-Completion (TC) diagram* which involves time relationship between the two manipulators was used. Over the years many other researchers developed this solution, extending it with a more complex environment. In 2014, Afaghani *et al.* [1] proposed an improvement of this method adding an SSV robot modelling which extends the original model and can be adopted to a wider category of robots. A comparison with the original algorithm performance is also shown.

Another way to solve the problem is to map the colliding obstacles into the joint space of the robot, as Park *et al.* did in [7]. In this way it would be easy to determine a collision-free trajectory just connecting the starting and the ending joint configurations with a broken line. However this solution is suitable only for robots with few degrees of freedom, as the problem complexity grows exponentially with it, and by the way the joint-space obstacle description is not simple at all even if the other obstacle is steady.

In order to solve collision avoidance problem, a feasible solution is made by *Artificial Potential Functions*, which introduce a dummy potential field which has to attract the robot arms to places where there are no obstacles and to repulse them when they are getting close to the obstacles [3] [8]. The main drawback is that the robot could move through a local minima of this potential and the algorithm can remain stuck. There are a few solution to this, depending on the number of degrees of freedom of the problem.

Part of our work is an extension to what it has been done in [10], where different types of robot modelling have been reviewed, including *Bounding Volumes* and *Point Cloud* modelling. While the experimental part was mainly focused on collision with a static generic object, our aim is to extend it to detect collisions with another robot. Since we have a lot of more information, our method must exploit it and so part of the modelling will be developed in a more efficient way.

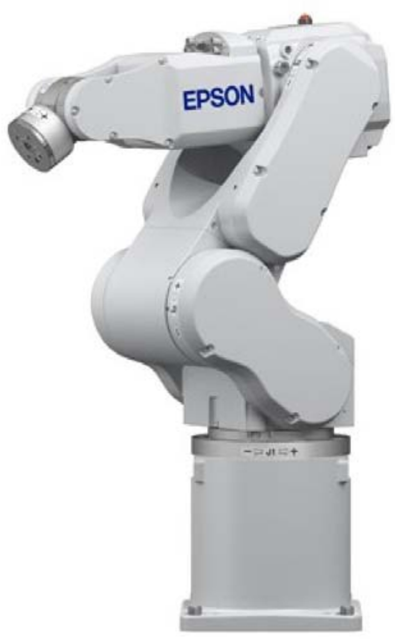
In this work we are exploiting the SSV modelling as proposed in [1] and the Bounding Volumes technique developed in [10], combining the fastness of the former and the reliability of the latter. Since we will be dealing with no time relationship between the robot trajectories, we developed an optimization of the algorithm which has the aim of excluding all situations which aren't harmful prior to the collision detection.

1.2 Environmental background

In our work we are dealing with mainly two types of robot manipulators, which are a 6-axes anthropomorphous and a SCARA (4 axes), but our procedure can easily be extended to other robots. The workcell we are dealing with may contain other objects, such as cameras, feeders etc., but we are focusing only in the interaction between manipulators. In this sense we are adding new features to what has been done before (cfr. [10]), so all other collision checks can be handled in parallel or prior to our phase, making our work almost independent to what has yet been implemented.

Each robot is provided with a controller, which purpose is to operate the movement of each joint of the manipulator and guarantee to follow a reference signal which is produced according to a higher-level movement command. In this sense the controller represents an interface between two separate abstraction levels, the *physical level*, which includes all the electric drivers and all the interchange signals used for the control phase, and the *end-user level*. What the end-user has at its disposition is a particular set of instructions (depending on the robot manufacturer) which are interpreted by the controller and then converted into reference signals for each driver. The biggest drawback during the research of a suitable method, due to the problem formulation, is exactly this, that is, we can operate only at user level and so we cannot act directly on the driver nor we can get directly any information from the field as we must query all the information by the interface provided. As a consequence, we cannot take into account any feedback from the robot and we can operate any trajectory modification only during specified times, since the robot can be stopped just after the execution of a single movement is completed. Moreover, we can modify the robot trajectory just adding new via points, that is, we cannot project the entire path. Even so, it might not be the best trajectory, as the time required to travel it could be very long since it is the result of a kinematic chain.

On our side we are provided with a tool called *Sequencer*, which is at a higher abstraction level compared to the end-user level. This has been developed in previous works by Robotics team at Università degli Studi di Padova. In this occasion we are allowed to work at this abstraction level, where we can get all informations about workcell and robot configurations. The aim of the sequencer is to handle the communication through the entire workcell, which involves interaction with cameras, sensors, feeders and all the other equipment, and then design the entire work cycle, which includes robot trajectories. At this level, the locations for each robot movement are generated and then they are passed to the controller as sequence of elementary movements. An advantage of this technique is that it can be made independent of the particular interface due to any different manufacturer.


(a) Epson® C4-A601S 6-axes manipulator

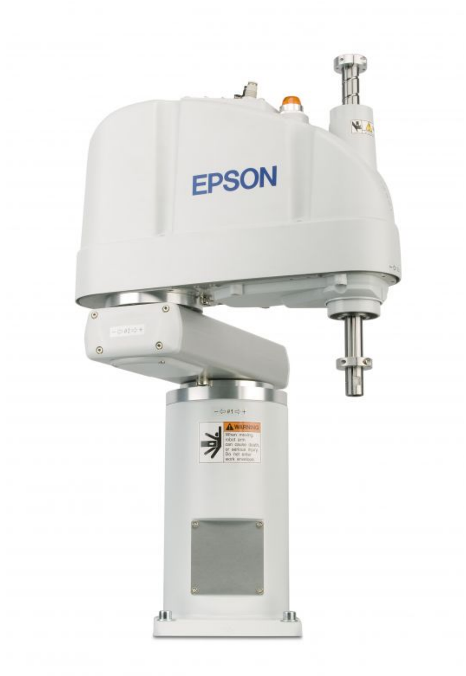

(b) Epson® G6-451S SCARA manipulator

Figure 1.1: Two examples of Epson® robots [source: Epson® site].

Despite this, we will be dealing only with Epson® robot manipulators (fig. 1.1), from which all the simulations and the analysis are obtained. Epson® interface provides the user with a simulator and a set of instructions, which can be divided into many categories. The main one of our interest is the set of *Robot Control Commands*[1], which includes all instructions for robot movements. The sequencer eventually has to generate elementary instructions which are compatible with this instruction set, so we must take into account this fact. The starting point is to analyse `Jump` instruction, which is the most used. We will explain it in detail in Chapter 2.

On the other side both the sequencer and all the analysis are developed in MATLAB® . Some critical phases are implemented with MEX functions, which core is C++, to exploit its fastness and task parallelism which is very frequent in our method. In some cases we provided also Matlab-developed GUIs to integrate user control during some configuration tasks.

Our work is organised as follow.

Chapter 2 has the aim of analysing true robot trajectories and finding an approximation of them which is suitable for the collision detection phase, within a certain tolerance.

On Chapter 3 we are providing a procedure to find whether two robots are colliding, given two sets of joint positions which are supposed to be known a priori.

On Chapter 4 we analyse the collision detection problem more deeply, providing an optimized algorithm which has the aim of improving the algorithm developed in Chapter 3 by making it real-time executable.

Chapter 5 contains a series of experimental results and some comparisons between different configurations available to the user.

Finally, some additional details to theoretical aspects and basic knowledge are summarized in Appendix.

1 for a complete instructions set, see `http://robots.epson.com/product-detail/168`

2 Trajectory Estimation

In this part we are trying to find a method which allows to represent a robot trajectory given the initial and final configuration and a set of via points. As we will see in Chapter 3, the trajectory must be presented as a sequence of joint positions, which must resume it within a certain tolerance. This is the result we shall reach after this phase.

For the sake of clarity, here the multi-robot environment is not needed as this shall be a general procedure and all the tests have been made on a single-robot environment.

Despite we tried to find the most possible general way, all the analysis part has been developed with an Epson® C4A601S anthropomorphous robot. With some modifications and different tuning, the procedure may be applied to other robots.

The first thing to point out is that we are dealing with two possible representations of the robot in space, which are the *joint space* and the *operative* or *cartesian space*. We assume the reader has some familiarity with these conventions, otherwise see [4]. Usually in a simple single-robot environment the goal is to move the end effector from one configuration (position and orientation) to another one, regardless of the other joints configuration. In this occasion the position of all the other joints are important since each link of the robot may collide with the other robot. For this reason we have to monitor not only the position of the end effector but the other joints as well. The easiest way to do this is using joint coordinates and the *kinematic model*, which allows to map joint coordinates into cartesian space. What matters is that once we know the joint configuration, we can know the position of each link in Cartesian space and therefore the obstruction of the entire robot, given the link shape description in each joint reference frame. For more details about the kinematic model, see Appendix C.

To get to the final result we tried different ways and came up with some possible solutions. Firstly, it is necessary to explain in details the environment where we are operating.

2.1 Problem statement

At this stage we are assuming to know an initial position[1], a goal position and some via points. The aim is to find how these points are connected over time. This depends on two main factors:

1 it could be known both in joint space or in cartesian space, the transformation between them is made by the *direct* and *inverse* kinematics, assuming we know one of the possible robot configuration when doing the inverse kinematics.

- ⋄ the type of instruction (e. g. `Jump3`, `Move` or `Go`);
- ⋄ the type of junction within adjacent movements (e. g. *Continuous Path* option presence).

Most of the following information are retrievable from the Epson® user manual. We summarize only the essential information which are needed for the next phase.

2.1.1 The Go instruction

This instruction moves the arm between the current and a specified position with a *Point-To-Point* (PTP) motion. This is a particular type of motion which is driven directly in joint space, which ensures the fastest movement between two points. There are a few options attached with this instruction, including CP motion (see Section 2.1.3) and parallel task execution, which cannot incorporate any movement instruction but is still useful for measurement acquiring. From now on in this subsection we assume CP motion is not enabled.

PTP motion planning parameters

Usually, once joint configurations $\mathbf{q}_0$ and $\mathbf{q}_f$ are known, there can be different laws which perform PTP motion, all differing in simplicity, differentiability and regularity of the motion itself. One of the most used in this field is the *trapezoidal-speed law*, which consists of a first *acceleration phase*, a central *cruise phase* performed at constant speed and a final *deceleration phase* (fig. 2.1). To completely determine the law some

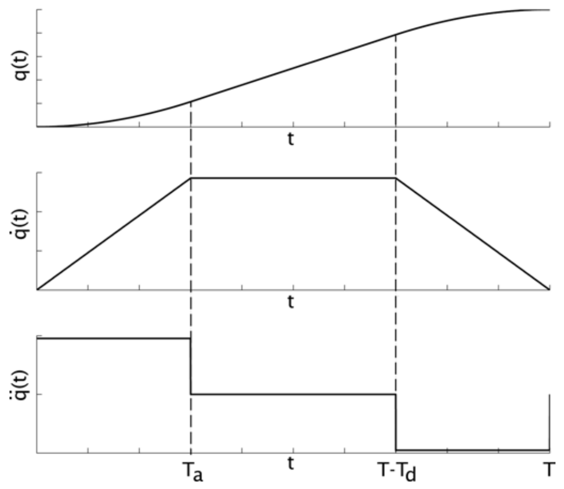


Figure 2.1: Trapezoidal-speed law time VS position, speed and acceleration graph.

constraints must be imposed, which act like parameters to be tuned in an identification manner. Some parameters are fundamental and will be reprised further on:

- ⋄ T_a is called *acceleration time* and it is the time travelled when the motion is uniformly accelerated, with acceleration a.
- ⋄ analogously T_d is called *deceleration time* and the motion during this time is uniformly decelerated, with acceleration d.

- ◇ T_c is the *cruise time* and it is travelled with constant speed (called *cruise speed*, V_{cr}) and consequently no acceleration.
- ◇ $T = T_a + T_c + T_d$ is the *drive time*.

2.1.1.1 *Experimental observations about timings in PTP motion*

With reference to fig. 2.1, we will assume to know initial and final positions[2] $\mathbf{q}_0$ and $\mathbf{q}_f$. The user, with a specific instruction, can set the *maximum allowed speed* and *acceleration factors* for PTP motion respect to *nominal maximum speed and acceleration*. These two values must be treated differently.

The speed factor doesn't mean that each joint is moving with the imposed speed, but that one of the joints is moving with its maximum speed allowed. This is because we have n (e. g., 6, if we are considering a 6-axes robot) laws like the one depicted in fig. 2.1 and, after some experiments, we discovered that the drive time T, the acceleration and deceleration time (T_a,T_d) are the same for all joints. In this sense the motion law is *synchronous*. To maintain this constraint it is logical that not every joint is moving with its maximum speed allowed, as long as the travel distance $\Delta\mathbf{q} = \mathbf{q}_2 - \mathbf{q}_1$ is different for every joint. A simple example is given by a null travel distance for a specific joint: whatever is the scale factor, the joint will maintain its speed to 0 through all drive time.

On the other hand, the acceleration factor depends on the maximum acceleration available, which relies on many practical conditions, such as robot current position, load weight and eccentricity. Hence the available acceleration is never known with precision, unlike the speed.

If we suppose the profile to be symmetric (i. e., $T_a = T_d$ and $a = d$) and for a moment we forget about the synchronisation between the joints, using law obtained in Appendix A and after some algebraic steps we would get that

$$T_a^{(i)} = \frac{TV_{cr}^{(i)} - \Delta q_i}{V_{cr}^{(i)}} \tag{2.1}$$

which holds for every joint i. A particular equation in (2.1) is the one which gives the most restrictive condition once we impose

$$V_{cr}^{(i)} = \alpha_v V_{max}^{(i)}$$

where α_v is the speed factor and $V_{max}^{(i)}$ is the nominal maximum speed allowed for current joint i. The acceleration time deduced from (2.1) is the time required to reach the cruise speed once boundary conditions (i. e., drive time, positions and acceleration) have been imposed. First thing to notice is that $T_a^{(i)}$ could be out of the range $\left[0, \frac{T}{2}\right]$ for some joints, in which case the cruise speed is not even reached for them. In this situation the trapezoidal law is degenerated onto a triangular law where $T_a = T_d = \frac{T}{2}$ and the overall acceleration time for all joints is $T_a = \frac{T}{2}$.

If $T_a^{(i)} \in \left[0, \frac{T}{2}\right]$ $\forall i$, instead, there exists at least one joint which reaches its maximum speed $V_{cr}^{(i)}$. If now we reintroduce synchronization between joints, $T_a^{(i)}$ in (2.1)

2 The graph in fig. 2.1 represents only the relationship for one joint

becomes a dummy time. After some simple steps it can be shown that the dummy speed obtained extracting $V_{cr}^{(i)}$ in (2.1) is

$$\dot{q}_i\left(T_a^{(i)}\right) = \frac{\Delta q_i}{T - T_a^{(i)}} \tag{2.2}$$

The *critical joint* is the one for which $\dot{q}_i\left(T_a^{(i)}\right) = V_{cr}^{(i)}$ and consequently $T_a = T_a^{(i)}$, which represents the most restrictive condition in (2.1).

At this point, once we know the drive time T and the scale factor α_v we know the entire law which rules the PTP command with CP off. The only problem is the knowledge of T. Some experimentations made with the Epson® C4A601S confirmed that this is the right way the controller is designing the trajectory for a PTP command with CP disabled.

Final data needed to solve the problem

To solve this problem in its entirety we need another information, which can be either the acceleration adopted for the critical joint or the drive time T. Fortunately we can retrieve T from the robot controller even offline, as soon as we know the joint coordinates of the endpoints of the path. This is performed by the instruction `PTPtime` in our case. Unfortunately to get this we have to query the controller from the sequencer and this may be a problem since it represents a loss of time. Another problem,as said, is that the acceleration available to the robot for each movement not only depends on the value imposed by the user but mostly depends on the robot joint displacement, the load weight and eccentricity. As a result, we can't take into account of a precise value of acceleration available constantly and therefore we shall find another way to solve our problem.

2.1.2 The `Jump3` instruction

This instruction is also known as *3D gate motion* and combines three stages: two linear motions and one PTP motion. The linear motion is performed along a straight line in Cartesian space. As a consequence, it is slower than the equivalent PTP motion between the same places, but it is useful in certain situation. To perform `Jump3` instruction we need to provide 3 points (P_2, P_3, P_4). The result is that from P_1 (the current point) to P_2 the motion is linear, then from P_2 to P_3 it is a PTP motion and finally P_3 to P_4 it is a linear motion again. The former stage is called *depart*, the middle one is the *span* and the latter one is *approach* stage. It is possible to skip the approach or the depart phase, which is usually done when we want to perform a gate motion with a certain number of via points, using an initial `Jump3` without approach, a sequence of `Go` and a final `Jump3` without depart. This is the typical situation we are going to study.

There are a few options available for the `Jump3` instruction, including the same for `Go`. However, here there is another important option that will affect significantly the trajectory.

2.1.2.1 *The `Arch` option*

The `Arch` option anticipates the beginning of the *span* motion in a quantity specified by the user, which is arch *upward* for depart motion and arch *downward* for the approach stage. The result is that the linear and the PTP motions are joined smoothly (fig. 2.2).

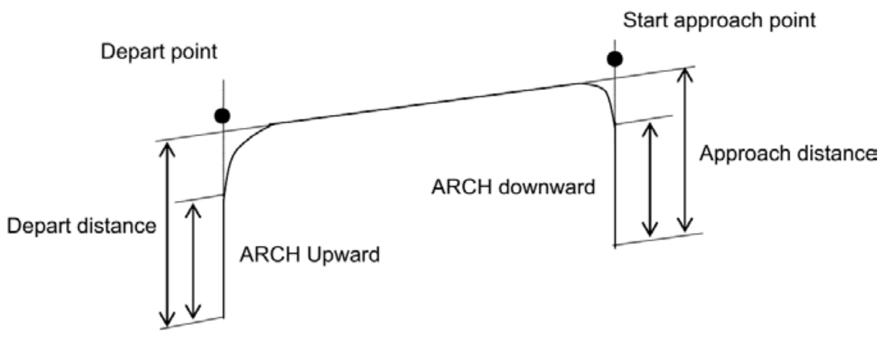


Figure 2.2: `Jump3` path with `Arch` option enabled [source: Epson® User Manual].

This option prevents the robot to stop when it would reach P_2 and P_3 and consequently the overall movement becomes faster. This, however, adds a new uncertainty to our identification problem, as we don't know when the junctions are going to rejoin the original span trajectory, nor we know the timings of these junction phases. This is another problem we had to cope with, considering that practical motions will almost always include the `Arch` option enabled.

2.1.3 The Continuous Path (CP) command

This option is available only for a restricted set of instructions. The typical situation when this is used is when a trajectory includes a via point. When this option is enabled, the motion which would start from the via point is anticipated in time and it starts when the previous motion begins its deceleration phase.

The visible result is that the trajectory legs are joined together in a smooth way and the robot does not stop at via point reach. Moreover, crossing of via point is not guaranteed and the point where the junction begins and ends up depends on the speed of each joint.

For each single joint, what happens is visible in fig. 2.3. To be able to preview the junction behaviour it would be necessary to know both the deceleration and the acceleration available in the middle phase. If CP option is enabled between two `Go` instructions, the acceleration which could be evaluated in Section 2.1.1 may not be the correct one since it supposes the endpoint speed is null, whereas in this case this is not true. Of course, the controller knows exactly which is the maximum acceleration available for every position, but this information remains hidden. For this reason, we have to find a valid approximation of the trajectory near via points.

Summarizing, there are two main problems we want to solve:

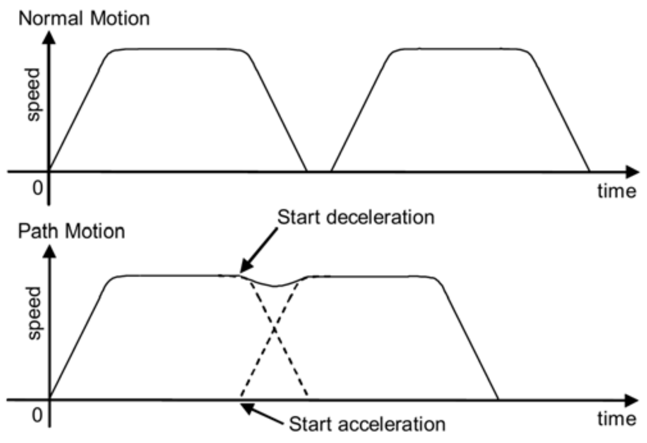


Figure 2.3: Comparison between movements with CP on and off for a single joint trajectory. If the CP option is not enabled, the robot reaches the via point and it stops there before continuing with the second leg (as it is on the above graph). With CP option engaged, instead, each engine is driven before, so the end effector is moving continuously [source: Epson® User Manual].

- ◇ find how `Arch` option joins the linear and the PTP motion and find an approximation of it;
- ◇ find how two PTP motions are joined together when `CP` option is engaged.

Despite seeming similar, the problem are totally different, as in the first case we have an additional information, which is the arch upward and downward, but we have to cope with a motion which isn't simple as seen in joint space (the linear one), whereas in the second case we have no information about the starting and the ending point of the junction.

We will analyse the problems separately, even though part of the solution is common between them.

2.2 Approximation of a `Jump3` motion with `Arch` option

Path and trajectory definitions

From now on, we will refer to *path* as the physical sequence of space positions, and the *trajectory* will be the path ruled with a time law. Moreover, the *reference path* is the path which is followed without `arch` nor `CP` option enabled.

For simplicity, we refer to a depart motion, since the processing for the approach motion is analogous.

2.2.1 Estimation of continuous trajectory

The first approach we tried was to find any time relationship which ruled the junction phase, that is, we wanted to find at which time the trajectory was re-joining the reference one. To completely describe the approximated trajectory we need to know the point where the reference trajectory is re-joined.The only information available is that the initial trajectory is the same (fig. 2.4), so we can suppose to know the starting point[3] P_a and the starting time t_0 when P_a is reached.

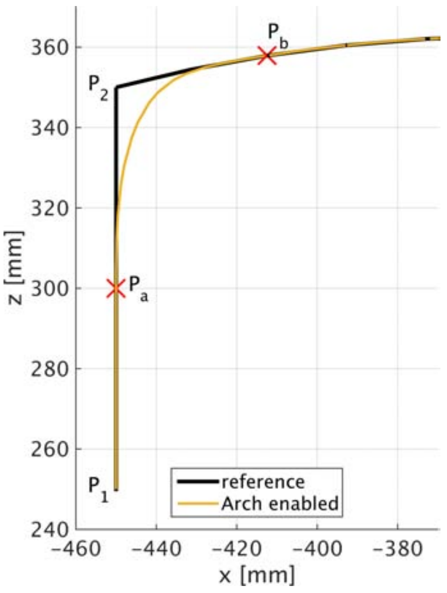


Figure 2.4: Path followed by the end effector during an Arch motion without appro. Points considered are highlighted. Notice that the path followed from P_1 to P_a is the same as the reference one.

Trajectory and path estimation trials

After some trials, we couldn't find any valid time relationship across a set of sample trajectories. This is due to the lack of information we have, including the fact that we are joining one PTP motion with one CP motion and the way how they are joined is not clear, plus any junction time law may be different between each joint (i. e., they could even not be synchronous). Even if we could find the right time law, this should be related to a specific final position, which has to be estimated as well.

We decided then to temporarily abandon the trajectory idea and instead we analysed only the path followed by the end-effector. Despite this, the problem to find point P_b remains opened. As this is an approximated procedure, we must define a tolerance within we can achieve the reference path. A first trial was to impose a specific position and then see how close it was to the real path, varying the speed factor, without temporarily consider the orientation (fig. 2.5).

Of course, the point imposed must be dependent on the specific path and on Arch settings. Let d_D be the Cartesian distance between P_a and P_2. For this phase we always imposed that the speed and acceleration factors were the same both for PTP and CP motion, even though different values can be set.

First of all, we imposed P_b to be the point on the reference trajectory which is exactly at Cartesian distance d_D from P_2 in the forward direction (otherwise it would be P_a again, see fig. 2.4). However, this choice was good only for certain speeds, whereas for other ones the junction point was reached before (fig. 2.5). The reason why the trajectory follows different routes depending on the speed is due to the physical limits of each motor, for instance torque limits, which are most likely reached at

3 without loss of generality, we assume that every time it is possible to convert a position between joint and Cartesian space.

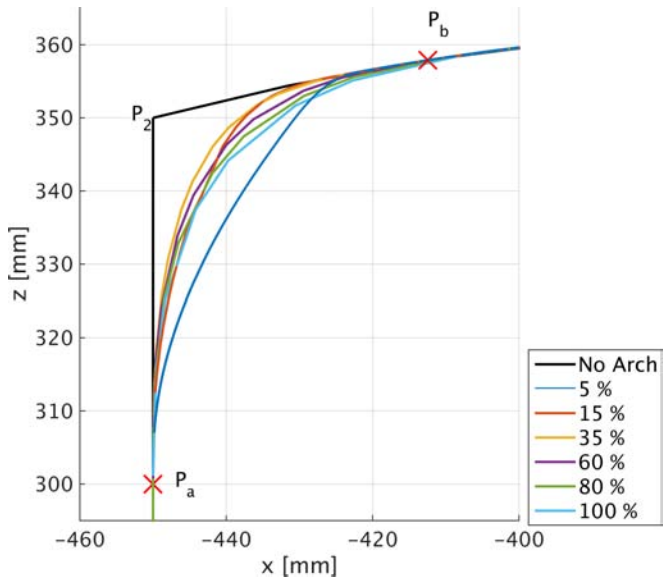


Figure 2.5: Path followed by the end-effector of the robot in function of the speed factor. This is an actual acquired robot trajectory (xz view only). Point P_b is the one which distance from P_2 is exactly the distance from P_2 to P_a, which is known a priori. The reference *path*, instead, is always the same regardless of the speed (black line).

high speed. However, since we don't have any precise information about the acceleration available at each moment, we cannot take into account of these limits.

Since we are mostly interested into high speed trajectories, we gave the priority on this feature and so we tried to make P_b linearly dependent on speed. In fact in fig. 2.5 we can notice that, apart from very low speed, the faster is the robot, the further is P_b from P_2.

At this point we can assume to know with certain precision re-joining point P_b in Cartesian space. However the problem is still not solved, as we need to reconstruct the path between P_a and P_b.

Ideas for joining P_a *and* P_b

Without any time-related knowledge, the only thing we can try to do is to join P_a and P_b with a linear junction in joint space. However this leads to poor results, as it would mean to impose a linear relationship between time and joint position, which leads to a discontinuity yet on acceleration, which cannot happen in practice. Any other position law between those two points would require time dependency, which at this stage we don't have. In fact the critical information we miss is the time required to cover the junction. Since this information cannot be retrieved from the robot, after several attempts we gave up with this idea and we switched to another technique.

2.2.2 Approximation using knots smoothing

Let us move a step backward. Regardless of the strategy used to identify the final trajectory, our aim is to get a series of knots which discretize the path and represent it with suitable precision. Up to now the idea was to first of all determine the continuous trajectory and then to sample the final outcome. Since we did not get any feasible result we decided to do the reverse, that is, first of all we quantize the reference path and then we process the knots in order to approximate the real path. Even in this case the time relationship is not solvable, but at least we hope to get a valid path approximation.

2.2.2.1 *Uniform path sampling*

First of all, we need to get only a small set of points from the continuous reference path. The easiest thing to do is to determine the number of points which we want for each leg of the path and then uniformly divide the path leg travel according to the joint space displacement of each joint (fig. 2.6). In other words, for a certain leg of the path and for each joint we find the maximum and the minimum position value (which in case of a PTP motion, like the second leg of a `Jump3` instruction, are the first and the last position) and we divide the interval into N sub-intervals, where N remains a parameter to be chosen.

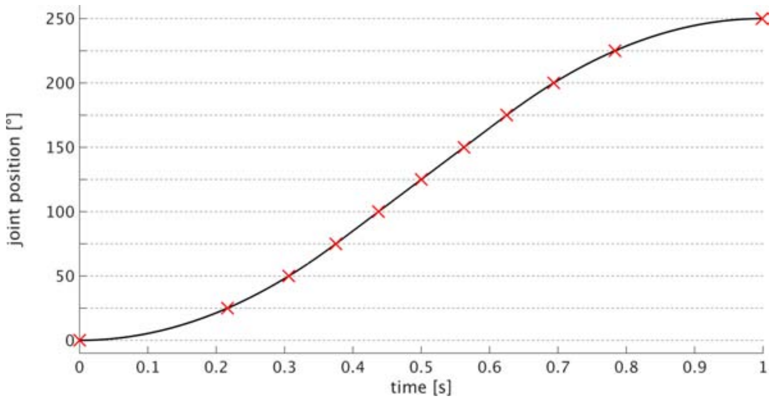


Figure 2.6: Uniform quantization of position trajectory followed by one motor, split into 10 sub-intervals. This is specifically for one joint, but it has to be repeated for all the others. The red crosses indicate the knots that will be used for the next phase. Notice that the subdivision is uniform along position but not along time.

As said, this technique is suitable only for the PTP motion. During a CP motion this method can still be used, but a better approach is to split the path according to the end-effector travel in Cartesian space. The joint subdivision is then obtained by performing inverse kinematics on the end-effector sampled position.

Suitable amount of knots to keep

At this level, the number of samples N to keep for each leg might seem not important, but, as we will discover in Chapter 3, the time required to the collision detection algorithm depends quadratically on the number of knots used for the entire path, so it advisable to keep as few knots as possible. On the other hand, the smaller is the set of samples from the continuous path, the poorer are the results as the gap between two consecutive knots is increasingly high. Moreover, the gap between knots directly affects the smoothing algorithms we will explain later on. The number of knots for each leg becomes then an important parameter which rules the performance of path estimation.

Since we need to know all joints obstruction, to get useful results we need to evaluate the position of each joint in Cartesian space[4]. At this point we suppose to know the sampled path in joint space, so we need to perform direct kinematics over the sub-chain of interest. For instance, the end-effector position is given iterating direct kinematics until the end, while to get joint 3 position we simply iterate it until $\mathbf{T}_{2a}$ is available. For details about direct kinematics, see Appendix C.

To improve the path reconstruction for an approach and a depart motion we decided to split the PTP motion in the `Jump3` instruction into 2 parts. The split point is the one for which the arch junction has rejoined the reference path (previously called P_b). The method explained in the previous section to get this point is a good starting point.

To cope with different arch values, we also split the depart motion of reference path into 2 parts, of which one is the arch upward (from P_1 to P_a) and the other one is the remaining leg until the target point (from P_a to P_2). In this way we can partition the first part of the PTP motion coherently with the last leg of the depart motion. On the approach phase an analogous method is followed.

2.2.2.2 *Smoothing algorithms*

Once the Cartesian reference path of each joint is sampled, we need to emulate the smoothing process done by the controller. In this section we are analysing the `Arch` option, but most of the steps can be extended to the junction between two PTP motions.

The idea is to consider the set of neighbours of the target point (P_2) knot and replace them with another set of knots which better approximate the real path (fig. 2.7). We will assume the knots are enumerated consecutively along the trajectory.

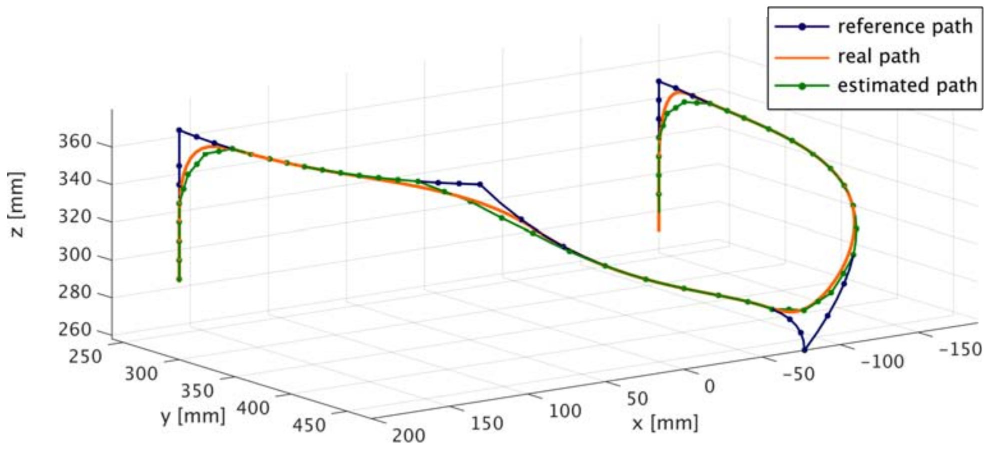
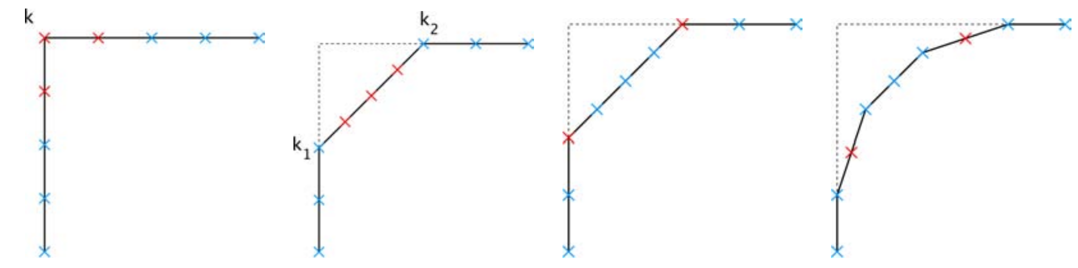


Figure 2.7: Sample path approximation with smoothing procedure. The blue line represents the reference path from which we applied the smoothing, resulted in the green line with relative highlighted knots. The real path followed by the robot is the orange one and it has been acquired from a real experimentation. The continuous line containing estimated knots is obtained interpolating knots in joint space with a first order interpolator (only for a clearer view).

4 with position of a joint in Cartesian space we mean the position of the origin of that precise joint, according to the Denavit-Hartenberg table

There are many ways in which this can be done. Two important informations that we have consider are the `Arch` upward and the speed factor, which can modify the smoothing process. These parameters are resumed into a *smoothing level* (l) which is fed into a specific algorithm. We will now outline some different algorithms which we designed to perform the smoothing process.

SYMMETRIC ALGORITHM The first algorithm we provided is simple. The set of knots which will be replaced is made of $2(l-1)+1$ adjacent knots centred around the knot correspondent to the target point (k).



(a) Step 1 **(b)** Step 2 **(c)** Step 3 **(d)** Steps 4 and 5

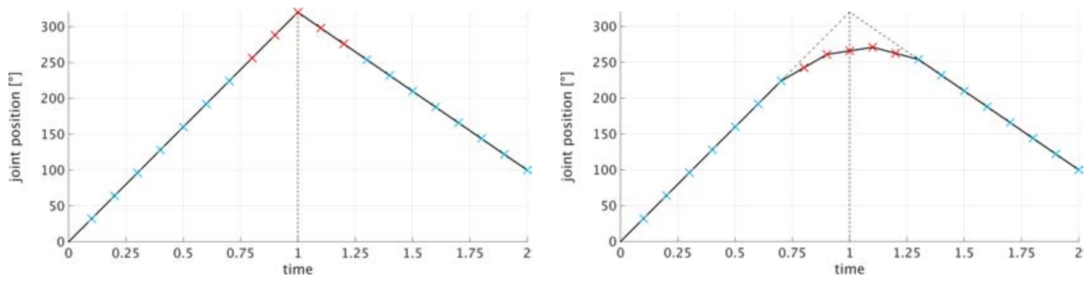
Figure 2.8: Steps for symmetric smoothing algorithm. Knots interested by each step are highlighted in red. Smoothing level in this example is set to 2.

The algorithm is resumed in the next steps:

1. remove $2(l-1)+1$ knots around target knot k (fig. 2.8a);
2. add $2(l-1)+1$ knots which uniformly split the interval (in joint space) between the two knots at the boundary of the deletion set (k_1 and k_2 in fig. 2.8b);
3. evaluate the offset $\delta = 2\left(l - \left\lfloor \frac{l}{2} \right\rfloor\right)$ to get to the knots which distance from k is exactly δ (fig. 2.8c);
4. position into the knot enumerated as $k - \left\lfloor \frac{l}{2} \right\rfloor - \delta$ and replace the next $\delta - 1$ knots with $\delta - 1$ knots which uniformly split the same interval (fig. 2.8d);
5. position into the knot enumerated as $k + \left\lfloor \frac{l}{2} \right\rfloor + \delta$ and replace the previous $\delta - 1$ knots with $\delta - 1$ knots which uniformly split the same interval (fig. 2.8d).

What we are doing with these steps is to anticipate the end of the joint movement and replace it with a linear junction in joint space (fig. 2.9).

ASYMMETRIC ALGORITHM This algorithm has the aim of emulating the smoothing made by the controller on approach and depart phases. The steps we outline are valid for a depart phase, while for an approach phase it has to performed with complementary operations. As the previous one, the amount of knots after processing is the same as the original path. The only input provided is the smoothing level, which has the same rule as explained for the symmetric one.

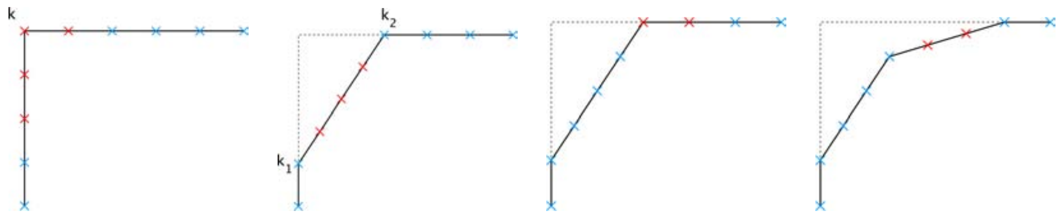


(a) Original path joint position. **(b)** Smoothed path joint position after processing.

Figure 2.9: Joint position trend before and after processing. In this case the underlying time is not physical time, as we do not know the time dependancy after processing. Hence the time distribution is not uniform and do not necessarily remain the same on the second leg after processing as we do not know how long the junction will last. However this is meant to show only the position behaviour, where the peak position is no more reached as the junction will anticipate the second movement.

1. Move to the knot labelled as $k - l - 1$ and delete the following $l + \lceil \frac{l}{2} \rceil$ knots (fig. 2.10a);
2. add $l + \lceil \frac{l}{2} \rceil$ knots which uniformly split the interval between the two knots at the boundary of the deletion set (k_1 and k_2 in fig. 2.10b). As the number of knots remains the same, associate deleted labels to new knots;
3. position in the knot marked as $k+1$ and replace the following l knots (fig. 2.10c) with l knots which uniformly split the same interval (fig. 2.10d).

The complementary algorithm is performed in a symmetric way and it is not reported here.



(a) Step 1 **(b)** Step 2 **(c)** Step 3 deletion set **(d)** Step 3 final result

Figure 2.10: Steps for asymmetric smoothing algorithm. Knots interested by each step are highlighted in red. Smoothing level in this example is set to 2.

ONE STEP SYMMETRIC ALGORITHM Another algorithm developed is a simplification of the symmetric algorithm which considers only first two steps. The result is the one depicted in fig. 2.8b. This can be useful sometimes when we want to mix different algorithms processing knots serially.

3-STEPS SYMMETRIC ALGORITHM Another variation of the symmetric algorithm is to add another step which smooths the path resulting as outcome near knot labelled k. The additional steps are

5. move to the knot enumerated as $k - \lfloor \frac{l}{2} \rfloor - \delta + 1$ and replace the next $\delta - 1$ knots (fig. 2.11a) with $\delta - 1$ knots which uniformly split the same interval (fig. 2.11b);
6. move to the knot enumerated as $k + \lfloor \frac{l}{2} \rfloor + \delta - 1$ and replace the previous $\delta - 1$ knots with $\delta - 1$ knots which uniformly split the same interval (fig. 2.11b).

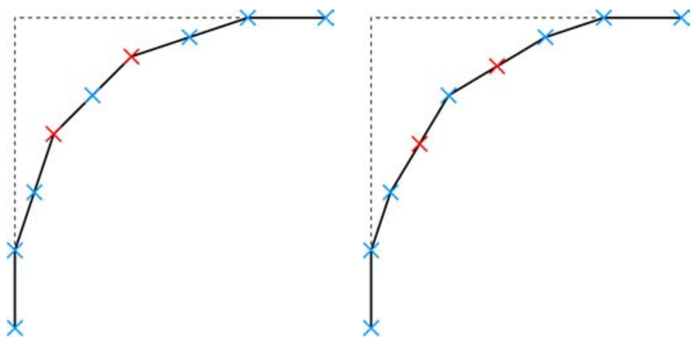

(a) Steps 5 and 6 (replaced set) **(b)** Steps 5 and 6 (final result)

Figure 2.11: Additional steps for symmetric smoothing algorithm which results in its 3 steps variation. Knots interested by each step are highlighted in red. Previous steps are depicted in fig. 2.8.

These are the main algorithms we used and during the experimental phase we found which one of these is most suitable for the `Jump3` motion and for PTP motion, in order to reduce the number of parameters to be tuned. In fact we would like a simple identification algorithm which is affected only by few parameters, otherwise we could result in an algorithm that depends too much on identification data set.

2.3 Approximation of the junction between two PTP motions

The approach to find the approximation when we want to join two PTP motions is pretty similar to what has been done before. However, there is a fundamental difference: we don't have any knowledge about the position nor the time when the junction is going to leave the reference trajectory, which makes this problem even harder to solve. On the plus side, the position trend on the two interested legs on the reference trajectory is known and quite simple, as it follows a trapezoidal-speed law. Moreover, Epson® suggests that the junction is going to start in coincidence with the beginning of the deceleration phase of the previous leg and it will rejoin the reference path at the end of the original acceleration phase in the next leg. However, time required to perform the junction is not known and neither are the maximum acceleration and deceleration available. So a continuous path estimation is not feasible with only these informations.

New problems from uniform path sampling

We then decided to follow the same approach as with the `Jump3` instruction. Most of the observations made in Section 2.2.2.1 are valid in this occasion as well. However there is a new problem: if we split the PTP path into uniform intervals, this will result in an interval width (i. e., knots distance) which depends on the total travelled length. At this stage this does not seem to produce any undesirable effect, but as soon as we have to apply any smoothing algorithm, this leads to a junction length which depends on the length travelled by each joint, which is a side effect that is not realistic.

PTP motion dependency on speed with CP on

There is another important drawback. In fact, similarly to what we showed on fig. 2.2, the junction length depends on the robot speed. In this occasion, we have experimented that the slower is the robot, the closer is the real trajectory to the reference one. If we do not take into account the time dependency, we cannot follow this trend, leading to poor results, as it turned out applying this method directly (fig. 2.12).

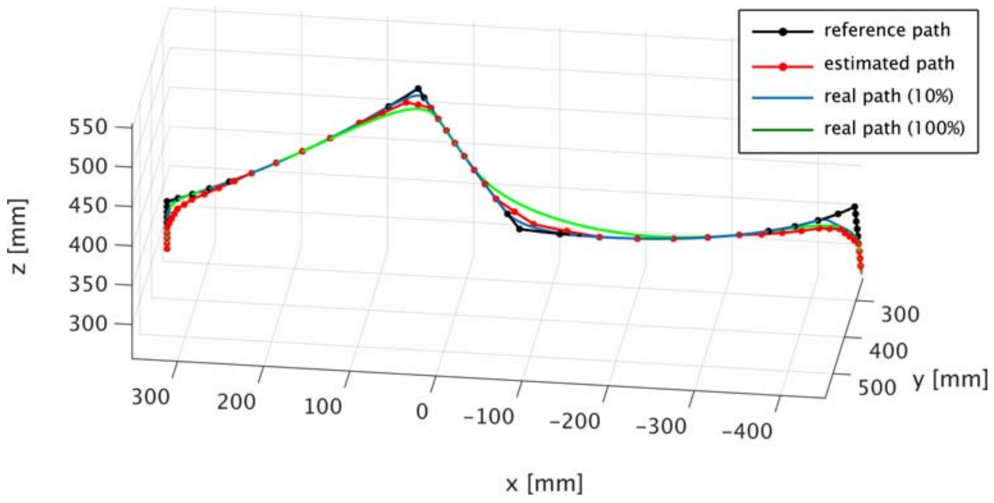


Figure 2.12: End-effector path with uniform quantization. The black line is the reference path, the red one is the estimated one and it is independent of the robot speed, while blue line is a real path obtained at low speed and green one is the same for full speed. As can be seen, the estimation is not suitable for all the possible speed factors. The problem is the same if we consider any other joint position.

2.3.1 Path sampling according to acceleration and deceleration times

If we want to improve the uniform sampling method we have to exploit any information available. All we know is related to the reference trajectory. If we are able to retrieve the drive time for each PTP motion with CP off, we can straightforwardly know the position at which each motor is starting and ending up its cruise phase. Let $\mathbf{P}_c$ and $\mathbf{P}_d$ be the position in joint space where each motor respectively starts its cruise and deceleration phase. For every junction, $\mathbf{P}_c$ refers to the next leg, while $\mathbf{P}_d$ comes on the previous one. These position inherit their dependency from the speed factor as the maximum acceleration is going to reach its limit with the rising of the speed. As a consequence, the higher is the speed factor, the longer will be the acceleration and deceleration phase and the further will be $\mathbf{P}_d$ to $\mathbf{P}_c$[5].

Speed dependency integration

The first idea is then to split each PTP motion path into 3 parts correspondent to *acceleration*, *cruise* and *deceleration*. The number of knots N available for the whole leg will be split as well in the parts, N_a, N_c and N_d. Only at this point we can apply

5 Remember that $\mathbf{P}_c$ and $\mathbf{P}_d$ in question belong to different legs

a uniform subdivision of each sub-interval using only the knots available for that specific part. In this way we have solved both the problems raised with the direct application of the uniform sampling to the whole leg. In fact, the slower is the robot, the shorter are acceleration and deceleration phases and then the estimated path remains close to the reference one (fig. 2.13).

After the sampling phase it follows a smoothing phase. The procedure to perform this is the same as the one used with the `Jump3` instruction, as explained in Section 2.2.2.2. Clearly, which algorithm and smoothing level to choose are parameters to set up during the experimental phase, as well as the number of knots available for each leg.

Practical adjustments for speed dependency

This technique allowed to reach good results, combined with a good smoothing algorithm. However, there are some situations where the estimated path is far from the real one. After some experimental observations we realised that the junction does not always rejoin the original path in $\mathbf{P}_d$ and $\mathbf{P}_c$, but, depending on the ratio $\frac{T_a}{T}$ and $\frac{T_d}{T}$, $\mathbf{P}_c$ can be anticipated and $\mathbf{P}_d$ can be retarded. This usually happens when $\frac{T_a}{T} > \frac{1}{3}$ and $\frac{T_d}{T} > \frac{1}{3}$, which means that the junction cannot last too long[6]. An example is shown in fig. 2.13c, where the blue dot represents a via point where the estimated trajectory rejoins the reference one prior to the position estimated (the green point on its right), as $\frac{T_a}{T} > \frac{1}{3}$ in the next leg, indeed.

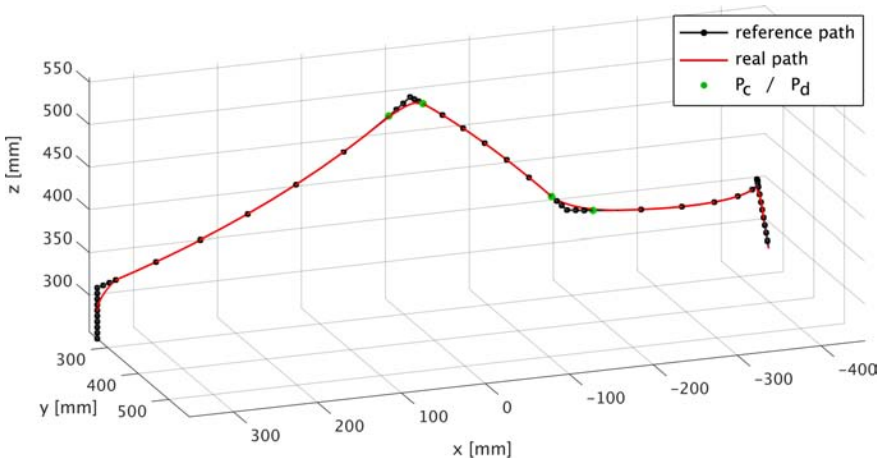
The previous condition happens when the speed is high, as said before. Hence we had to correct our estimation and impose to split each path leg either in $\mathbf{P}_c$ or in $\mathbf{q}_{\tilde{T}}$, where $\mathbf{q}_{\tilde{T}}$ corresponds to the point which is at $1/3$ of the time of the reference trajectory. The same reasoning is made for the deceleration phase. The condition to observe depends on the most restrictive one.

The main thing about this method is that it needs to know T_a and T_d from the reference trajectory, which can be onerous in some situations. Then we provided another method which tries to get to similar result without these quantities.

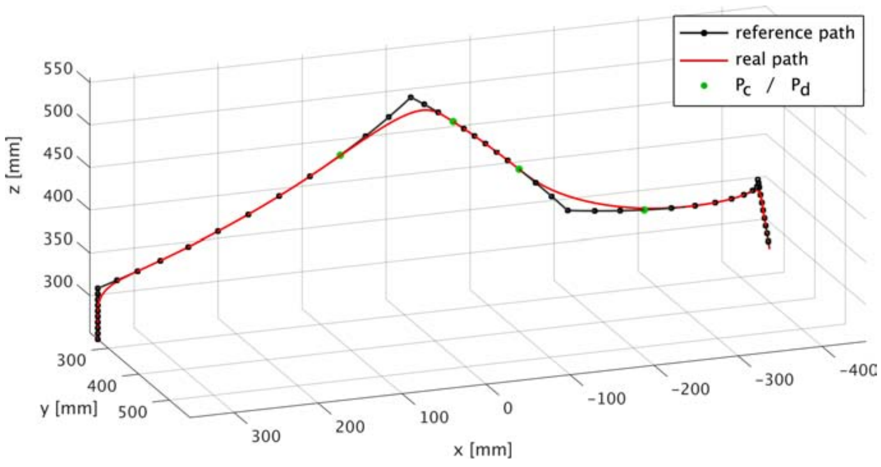
2.3.2 Path sampling without robot querying

Clearly, if we want to inherit speed dependency on sampling phase without any knowledge of T_a and T_d we have to accept less reliable results, as we need to approximate even more. To cope with this lack of information we decided to perform something similar to the correction we made on the previous section. Since we do not have the time dependency we apply that idea directly on the position of each joint. In other words, we compute the joint travel between the two extrema of each path leg and we force $\mathbf{P}_c$ to be at a certain fraction of the total travel, according to the speed factor. The same is done for $\mathbf{P}_d$. The mapping from speed factor to this coefficient is obtained according to experimentations made with a set of sample trajectories.

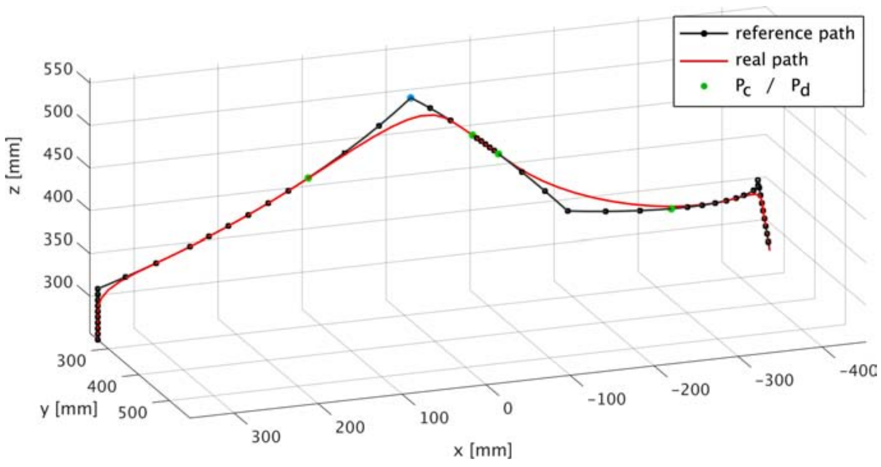
6 Note that if we impose the *acceleration coefficient* to be the same as the *deceleration coefficient*, the trapezoidal-speed law used by reference path is symmetric and this condition is simplified



(a) Speed factor set to 5%.



(b) Speed factor set to 50%.



(c) Speed factor set to 100%.

Figure 2.13: PTP motion estimation with path subdivision into acceleration, cruise and deceleration. Green points represent the boundary knots which divide these phases. It can be seen that the split points are moving according to the speed factor. Here only end-effector path is shown but this rule applies to all other joints.

Clearly, we have no warranty about the precision of our estimation, as we didn't with reference path timing knowledge, since both these methods are modified with correction factors obtained with only a set of possible situations.

However, to analyse performance of these methods we validated the estimation using another set of trajectories which realistically can be followed. This is explained in Section 2.4.

Why we did not follow a rigorous way

Finally, we want to recall that this method is only a partial work, as long as it is part of another thesis work, where it has been even improved. This is only meant to show some basic ideas about how to reach the goal, without being too rigorous on the final outcome. As we will see in Section 2.4, the goal is not reached for every trajectory we tried.

Another aspect which has not been faced is related to its practical implementation. The final purpose is to run this algorithm in real time. To do this, we need to get rid of any unnecessary operation, such as intermediate results and all plots and reports. This is the reason why there is no time-related performance analysis at the end of this phase.

2.3.3 Other possibilities and improvements

To cope with lack of a formal demonstration of the validity of this phase there is another improvement which can be performed with only a small computational additional cost. The main problem of the path estimation phase is that sometimes points

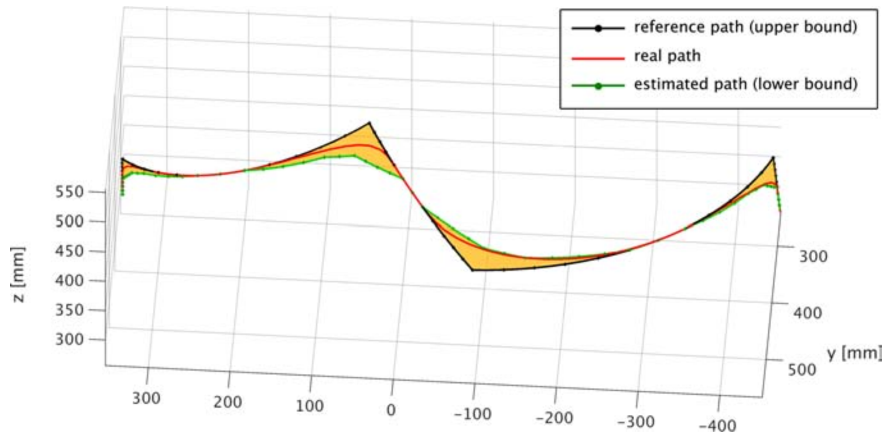


Figure 2.14: End-effector path estimation with two boundary paths. The black line is the reference path while the green line is the estimated one. The red trajectory is the real one and it is confined within these two.

$\mathbf{P}_c$ or $\mathbf{P}_d$ do not correspond to the points where the real path rejoins the reference one. Eventually we will be smoothing the original path around the via points (or the target points if we are in the approach or depart phase) providing only the smoothing level, which can produce very different results. However, if we are able to guarantee the real path to be within two estimated paths, we can solve the problem in a collision detection manner, as we are going to replicate it for all the possible trajectory combinations. Moreover, since most of the path is common, we can simply add dummy knots to the estimated path, thus reducing the computational cost of replicating the same algorithm. A conservative choice would be to consider a boundary path which corresponds to the reference trajectory and the other one as processed with the previous methods (fig. 2.14).

This, however, represents a future improvement and has not been implemented.

2.4 Experimental results on trajectory estimation

Here we summarize the main results obtained during the experimentation phase with the final parameters. We recall that these are only partial results.

Practical setup and requirements

First of all, target was to estimate the real trajectory within a tolerance set to $20\ mm$. Both the reference trajectory and the real one are acquired from a real robot movement. Robot in question in this phase is always an Epson® C4A601S with a $90\ mm$ length tool and no load within the gripper. To get the trajectory from the robot we launched a parallel task which has to continuously acquire robot position. Obviously the acquisition is sampled. Since the data is written into a local file and then transferred via Ethernet on a PC, we have a temporal constraint for the minimum sampling time, below which the results are unpredictable. As a result, we set the sampling time to $T_s = 10\ ms$. If the target is just the path and not the temporal distribution, we can also use the simulator. As it turned out, the reliability of the simulated path is good, whereas the time distribution depends also on the PC settings and it may vary significantly. In our case we can say that there are not big differences between the real and the simulated path because there is no load within the gripper of the robot. However, despite being possible to set load parameters (such as weight, inertia and eccentricity), simulation can vary slightly more in different situation, as the controller has a feedback from the real field. The differences can be even more evident on the dynamics, as for example the robot might encounter acceleration limit problems which are not found during simulation, due to the different real parameters of the load.

All datasets we used are composed by data acquired from a movement made of a first depart motion, a series of PTP motions (depending on the number of via points) and a final approach motion. This motion is repeated for different values of speed factor, which combines acceleration and speed available. Although Epson® instructions set allows to set a different speed and acceleration ratio both for a CP and a PTP motion, we decided to impose a general factor (previously called "speed" or "scale" factor) which we set to be the same for all these 4 parameters. This leads only to a subset of all the possible combinations, but it is the most interesting in our case. We must recall that the available acceleration is also dependent on the robot position and on the load parameters, whereas maximum speed available is known from Epson® manual.

Robot obstruction points

First of all we want to clarify which points of the robot are considered when we analyse the robot path. These are the origins of each Denavit-Hartenberg frame and

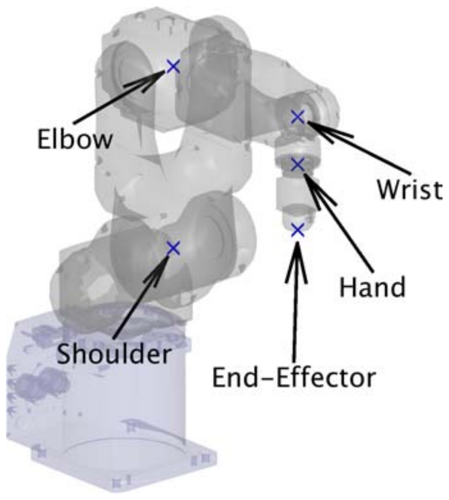


Figure 2.15: Critical points of an Epson® C4A601S robot which will be referred to during the analysis phase.

are depicted in fig. 2.15. As can be seen, not all critical points are interesting, as origins of joints 3 and 4 coincide, while origin of joint 1 never moves.

Since this is only partial results outline, the processing phase has been split in two different parts, as described in the previous section, which regards the approximation of the robot path near the target points (i. e., a `Jump3` motion) and the approximation of consecutive PTP motions between via points.

Parameters setup

Parameters used can be different as well. We report only the final values which can achieve the best results. All the analysis of the PTP motion has been performed with this setup:

- ◇ number of knots per leg including endpoints: 12 (i. e., 11 intervals);
- ◇ number of intervals for acceleration and deceleration: 3;
- ◇ number of intervals for cruise phase: 5;
- ◇ smoothing algorithm: Symmetric Algorithm;
- ◇ smoothing level applied: 2;

We can adapt the number of knots used per leg if the resulting knots for the whole trajectory are too much. We do this simply providing less knots to the cruise phase for each leg, which should preserve the maximum error obtained. Clearly, the first and the last leg belong to the Jump motion and so we have to skip the acceleration and deceleration phases for respectively PTP motions.

For the depart and approach motions the parameters employed are:

- ◇ number of intervals for straight legs: 9;
- ◇ number of intervals for arch upward and downward: 4;
- ◇ number of knots for PTP legs: 5;

◇ smoothing algorithm: Symmetric Algorithm;

◇ smoothing level applied: 3;

If, after the smoothing phase, there are too many knots, we can simply delete some of them where their density is greater. This is usually frequent at low speed, where there are a lot of knots near the via points and the target points. However this has not been done during this analysis and it can be a future improvement.

Results outline

A sample trajectory obtained with these parameters is depicted in fig. 2.16. The results obtained after processing these paths are instead in fig. 2.17.

To quantify the accuracy of the method we define the *estimation error* of a specific knot as the minimum Cartesian distance between that specific knot position and the real path in Cartesian space. Clearly we have to define an error for all paths followed by each critical point of the robot. The error between the original path and the real one is depicted in fig. 2.18, while the error obtained after processing is in fig. 2.19. Since the estimated trajectory time distribution is not known, the x axis in these figures represents only an identifier of the knots.

Tests with robot querying

We also did some tests with additional information coming from robot querying, as described in Section 2.3.1. Only PTP motion parts have been processed in this phase, as the approach and depart legs are not affected by this additional information.

We did tests with a large set of trajectories, of which we report only a resuming table that contains main results. In Table 1 different fields appear:

◇ ID: it is and identifier of the trajectory under consideration;

◇ scale: it is the scale factor of that particular trajectory;

◇ V: it is the number of via points in the trajectory;

◇ e_P^0: it is the position error without any processing, using the original path. This refers only to the PTP motion (the central one);

◇ e_J^0: it is the position error without any processing, using the original path. This refers only to the Jump motion (i. e., approach and depart motion);

◇ e^0: it is the position error without any processing, considering the whole path, therefore it is the maximum between the previous two fields;

◇ e_P^s: it is the estimation error after processing. This refers only to the PTP motion (the central one);

◇ e_J^s: it is the estimation error after processing. This refers only to the Jump motion (i. e., the approach and depart motion);

◇ e^s: it is the estimation error after processing, considering the whole path, therefore it is the maximum between the previous two fields. This result is achieved without robot querying;

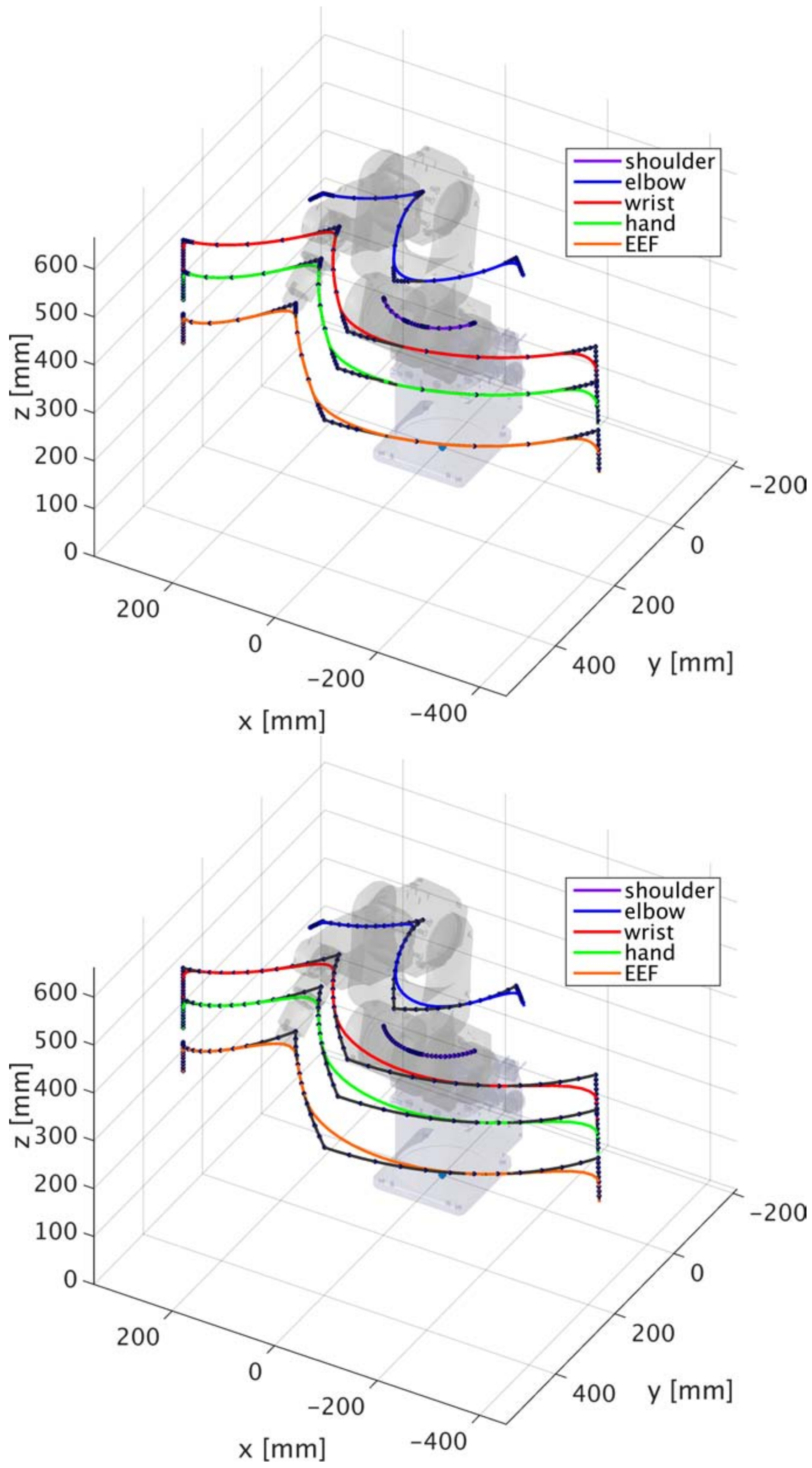
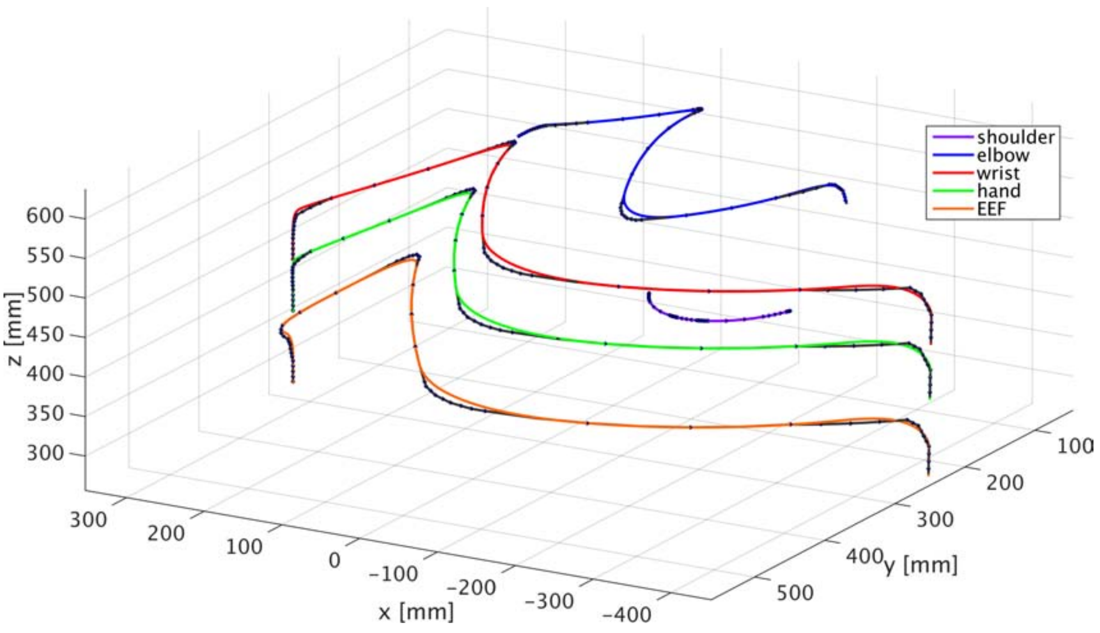


Figure 2.16: Trajectory followed by each robot critical point. The blue dotted line represents the reference path and the dots are the knots used for next smoothing phase. These results are obtained with a scale factor set to 25% (on the top) and at full speed (on the bottom).



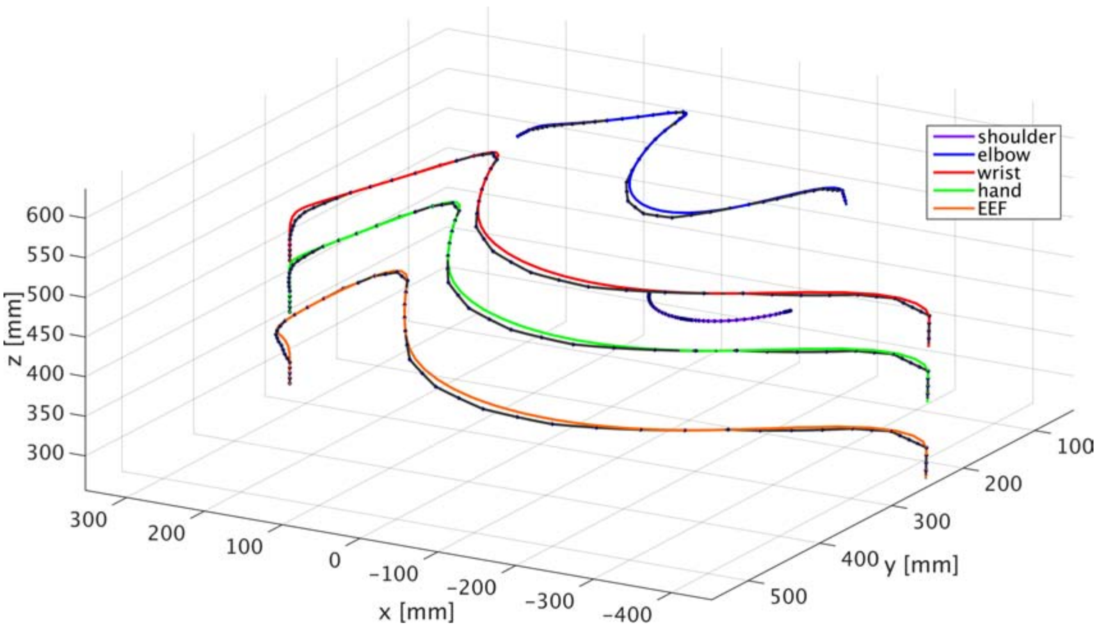
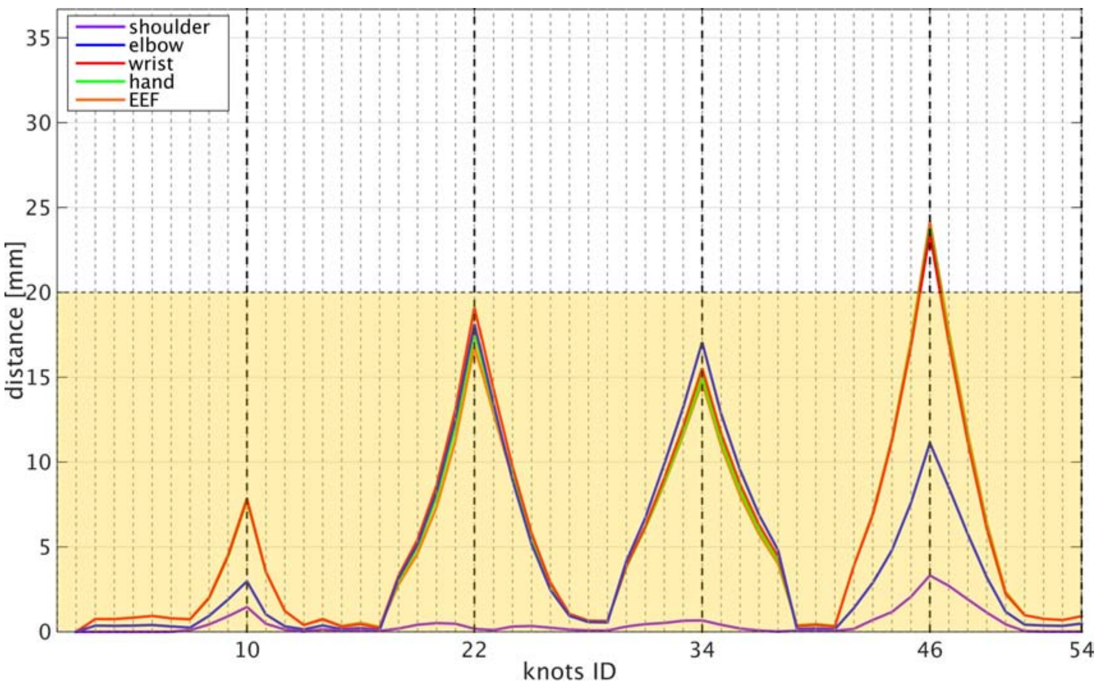


Figure 2.17: Trajectory followed by each robot critical point and its estimation after smoothing process without robot querying. The blue dotted line represents the estimated path. These results are obtained with a scale factor set to 25% (on the top) and at full speed (on the bottom).



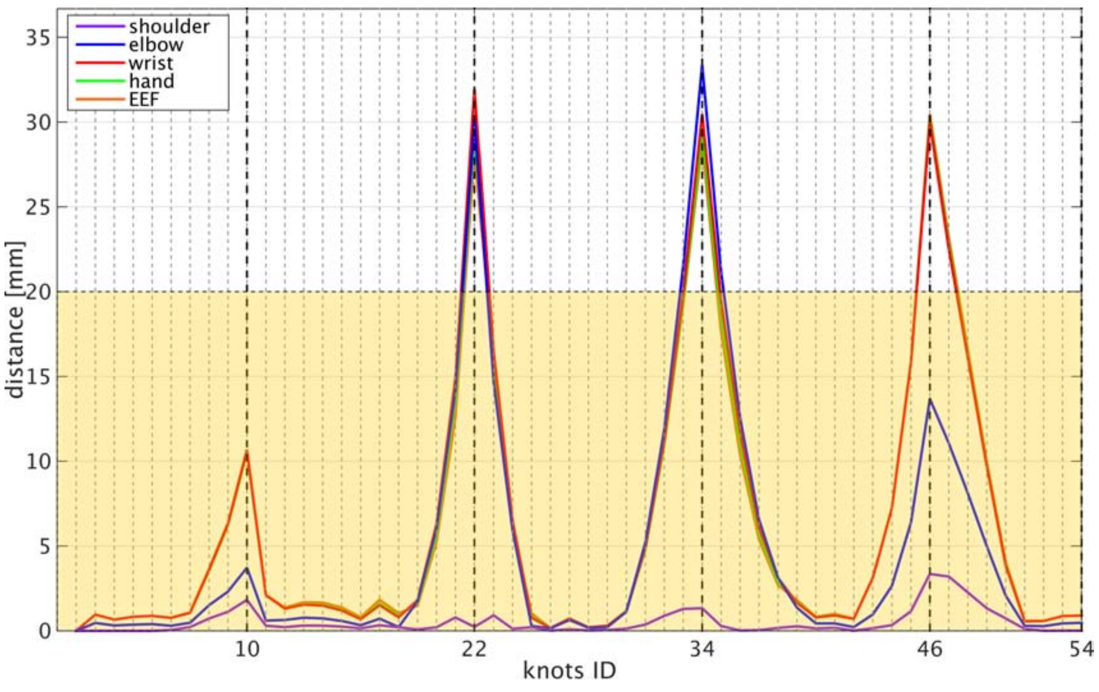
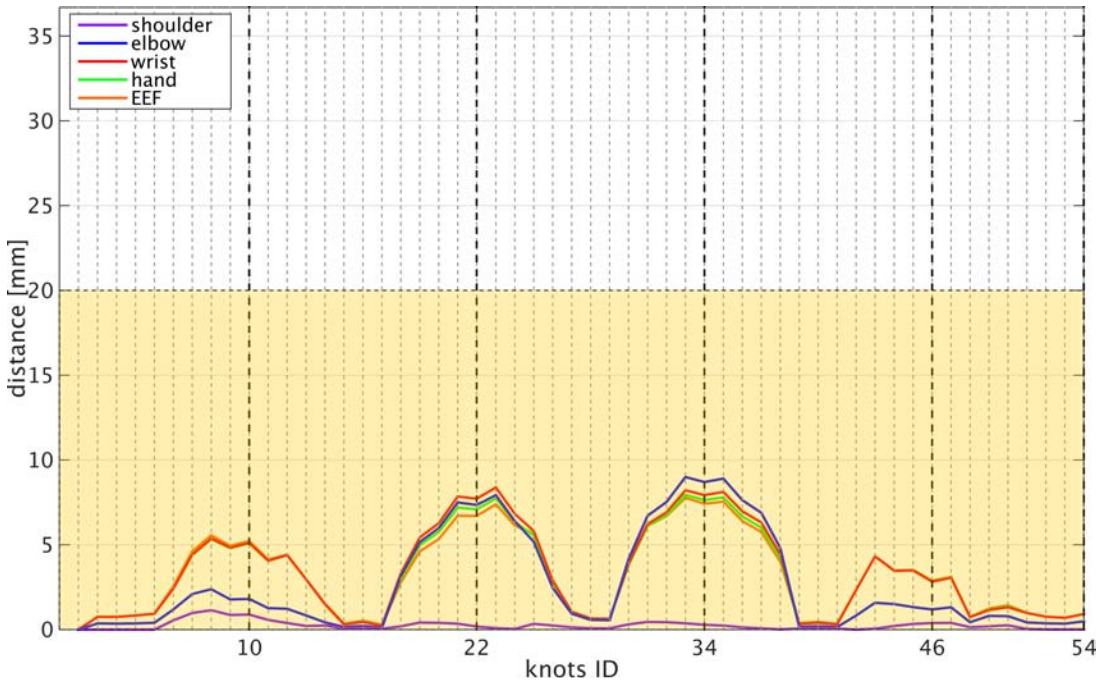


Figure 2.18: Position error using the original unprocessed path (cfr. fig. 2.16). These results are obtained with a scale factor set to 25% (on the top) and at full speed (on the bottom).



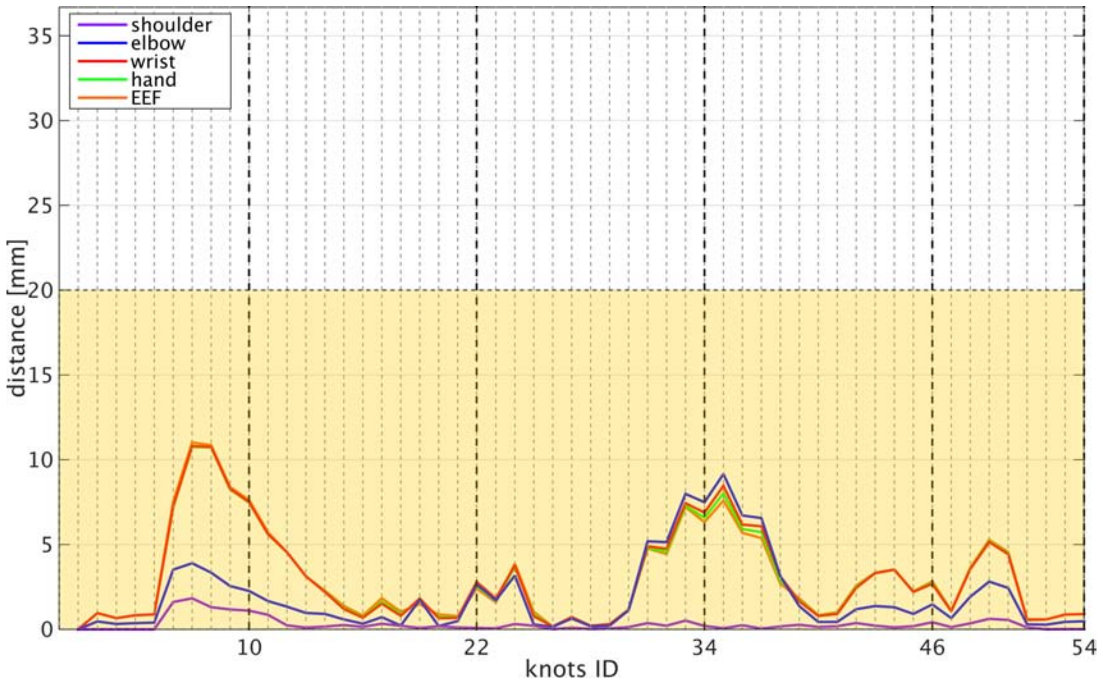


Figure 2.19: Estimation error after processing (cfr. fig. 2.17) without robot querying. These results are obtained with a scale factor set to 25% (on the top) and at full speed (on the bottom).

- e^T_P: it is the estimation error after processing, with additional information from the robot. This refers only to the PTP motion (the central one);
- e^T: it is the estimation error after processing with additional information from the robot, considering the whole path, therefore it is the maximum between the previous field and e^s_J, as the smoothing process for the Jump phase is identical.

Clearly, the errors reported are intended to be the maximum error across all the specific path. Since for each knot we have 5 different error definitions, the maximum value may have been found for a specific critical point of the robot. On this purpose on Table 1 we highlighted the error columns with different colors according to the critical point which reported the maximum error: purple values indicate the critical point is the wrist, blue one indicates the elbow and orange ones the end-effector. Red-filled cells indicate the results which are not within the threshold imposed as target.

Observations about final results

The first thing we may notice is that the critical joint is almost always the end-effector, as its position must be iterated through the entire kinematic chain. However, error trend of wrist and hand is quite similar to this, as long as the last three joints are imposing the arm orientation. In particular this is true for the end-effector, which position referred to the hand is always the same. First two joints rarely appear when considering estimation error, since they are less affected by smoothing process because their travel in Cartesian space is usually shorter than the last joints in the kinematic chain.

Another thing to notice is that, as expected, time knowledge usually leads to better results, as the resulting error is more often below threshold. By the way, parameters adjustment depending on speed allowed to reach good results even at high speeds, where the CP contribution leads to longer junctions (cfr. fig. 2.16). For the sake of clarity, on Table 1 only 4 different speed cases were reported, but this behaviour is found even with other speed factors.

Despite the goal is not always reached, there is a coherence outcome, as always there is a significant reduction of the error after processing.

We remark the fact that these are only partial results, so, even if the goal has not been reached for all trajectories, better results may be obtained improving this technique.

ID	scale [%]	V	e^o_P [mm]	e^o_J [mm]	e^o [mm]	e^s_P [mm]	e^s_J [mm]	e^s [mm]	e^T_P [mm]	e^T [mm]
1	10	3	38.0	34.7	38.0	10.5	10.0	10.5	9.8	10.0
2	10	2	45.6	50.7	50.7	26.3	24.5	26.3	28.5	28.5
3	10	2	27.3	26.5	27.3	11.7	8.4	11.7	5.7	8.4
4	10	3	18.2	17.3	18.2	4.9	4.8	4.9	4.9	4.9
5	10	2	38.4	37.6	38.4	8.8	16.0	16.0	13.4	16.0
6	10	3	35.4	43.0	43.0	27.4	13.2	27.4	9.7	13.2
7	10	3	47.2	46.3	47.2	24.9	21.3	24.9	10.4	21.3
8	10	2	15.6	15.2	15.6	6.1	6.9	6.9	6.6	6.9
9	10	2	40.3	39.4	40.3	28.6	21.1	28.6	25.3	25.3
10	10	2	33.2	32.4	33.2	13.8	16.0	16.0	20.1	20.1
11	10	3	32.1	20.1	32.1	25.0	6.8	25.0	6.9	6.9
12	10	2	18.7	22.5	22.5	12.0	13.6	13.6	12.2	13.6
1	50	3	80.7	28.4	80.7	11.3	10.9	11.3	17.1	17.1
2	50	2	45.8	44.9	45.8	15.2	21.1	21.1	20.8	21.1
3	50	2	27.6	26.8	27.6	10.6	8.9	10.6	6.9	8.9
4	50	3	19.9	19.0	19.9	8.0	5.4	8.0	10.1	10.1
5	50	2	37.8	37.0	37.8	11.5	15.2	15.2	9.1	15.2
6	50	3	41.9	33.9	41.9	21.7	4.4	21.7	8.2	8.2
7	50	3	48.2	47.3	48.2	18.5	22.3	22.3	5.2	22.3
8	50	2	17.3	16.8	17.3	5.8	4.2	5.8	5.9	5.9
9	50	2	40.4	39.5	40.4	24.5	22.1	24.5	19.0	22.1
10	50	2	33.9	33.1	33.9	14.3	17.8	17.8	15.8	17.8
11	50	3	39.9	19.6	39.9	22.1	6.2	22.1	9.7	9.7
12	50	2	37.1	20.0	37.1	22.0	16.4	22.0	14.8	16.4
1	80	3	94.4	30.7	94.4	15.4	12.0	15.4	11.7	12.0
2	80	2	45.5	44.6	45.5	18.9	21.1	21.1	14.6	21.1
3	80	2	29.5	28.6	29.5	9.8	11.7	11.7	7.3	11.7
4	80	3	19.5	18.5	19.5	11.7	5.2	11.7	10.7	10.7
5	80	2	40.1	39.3	40.1	15.8	18.6	18.6	9.2	18.6
6	80	3	39.0	38.2	39.0	7.7	8.7	8.7	6.1	8.7
7	80	3	51.8	50.9	51.8	10.2	27.0	27.0	8.1	27.0
8	80	2	19.9	19.4	19.9	6.7	3.1	6.7	4.8	4.8
9	80	2	40.1	39.2	40.1	20.0	21.5	21.5	18.3	21.5
10	80	2	32.2	31.3	32.2	25.9	15.3	25.9	8.4	15.3
11	80	3	40.9	21.8	40.9	12.5	8.8	12.5	8.6	8.8
12	80	2	38.4	21.4	38.4	15.1	12.6	15.1	13.5	13.5
1	100	3	108.8	30.5	108.8	21.6	8.9	21.6	12.2	12.2
2	100	2	41.8	42.8	42.8	25.6	18.1	25.6	12.7	18.1
3	100	2	33.3	29.6	33.3	9.2	13.0	13.0	11.4	13.0
4	100	3	17.4	16.6	17.4	16.1	4.7	16.1	14.9	14.9
5	100	2	52.1	40.3	52.1	18.5	20.1	20.1	9.0	20.1
6	100	3	37.1	40.1	40.1	6.7	11.2	11.2	5.6	11.2
7	100	3	53.4	52.4	53.4	9.7	28.8	28.8	11.0	28.8
8	100	2	21.6	21.0	21.6	7.3	3.6	7.3	4.7	4.7
10	100	2	29.3	28.5	29.3	36.9	11.8	36.9	19.4	19.4
12	100	2	38.0	23.3	38.0	14.6	11.0	14.6	12.7	12.7

Table 1: Resuming table with main results obtained during trajectory estimation phase.

3 Collision Detection

In this chapter we shall check whether the robot trajectory planned is free from collision with other obstacles. Here we must reintroduce the multi-robot environment. There are at least three kinds of obstacles which the robot could collide into. These can be:

- ◇ the robot itself, causing a *self collision*;
- ◇ another robot;
- ◇ a generic obstacle which isn't a robot, i.e. a *limit plane* or any pallet.

Self collisions

The first situation is not very common due to the limits superimposed by the controller. Typically this type of collision could involve the end effector and the first link. This situation can easily avoided because, unlike the other two, *self collisions* depend only on the joints configuration of the robot, the size of the tool, and the object carried by the end-effector, which are supposed to be known a priori.

Collision with another robot

The second situation is the one we are studying. The collision presence relies on both joint configurations and on the relative positions of each robot base. Once we know the exact trajectories of each robot we could ideally state if there is a collision in every moment. By the way practical implementations have some drawbacks:

- ◇ the trajectory is known only within a discrete set of times, that is, we can only test the presence of collision during precise moments in the trajectory and we don't know what happens from one moment to another one;
- ◇ the trajectory isn't the real one, but it is estimated, so it is affected by errors we must take into account;
- ◇ the robot shape isn't the real one, but it is simplified.

The main reason for these problems is computational cost. The smallest is the set of times we are using for each trajectory, the faster will be the detection, but the more inaccurate will be the result. On the other hand, the simpler is the shape used, the faster will be the detection, but the higher is the *false alarm rate*, which takes into account of false collision detections, due to the approximations of the robot shapes. We will discuss discuss this problem further on.

Collision with other obstacles

The third class of obstacles we can encounter is the most general and it has been studied in [10]. Since the situation can be very different, it must rely on the presence of other instruments capable of acquiring the shapes of the obstacles and their position, such as cameras, or on a previous workcell calibration and description.

In this work we will only focus on the second class of obstacles described and from now on we will assume to be in this situation.

Collision detection between robots

The aim of this phase is to decide whether the robot shapes are interpenetrating, in which case we will state that the robots are colliding. The problem is then reduced to the decision about the interpenetration between two solids, which represent and approximate the robot shape. This problem can be solved in different ways. In [10] a solution representing one robot with a solid and the other one using a set of points is presented. He then postponed the problem to the existence of any point inside a solid. The biggest drawback of this method is the right choice of the points of the second robot. A special algorithm was developed, which had the aim of correctly distribute these points around each robot surface. This combines accuracy and computational cost, which is proportional to the number of point chosen. This is not an equivalent condition for the collision presence, but in practical situations it is seen that this is met every time there is a collision and the probability of *false negative* [1] is negligible. Moreover, the point distribution remains the same in each moment, so it can be checked once offline. Anyway the time cost is still a critical problem.

Robot shape overview

We provide instead another method, which is in some sense directly descending from the original problem, that is, we directly consider the robot shapes. These depend strictly on the joint configuration of each robot. To compute the collision test, once we know each robot joint configuration, we would have to compute the shape of each manipulator and then apply a solid interpenetration algorithm to these shapes. Unfortunately, computing a solid which changes its shape at every moment is too onerous and moreover the interpenetration test wouldn't be too simple. In light of this we must find a way to test collisions which doesn't require to recompute the shape on-line, but only a rigid transformation (translation and rotation), which is faster to perform. The natural choice is then to consider a solid for each link of the robot, which does only translations and rotations referring to a static frame (i.e. the *station frame*).

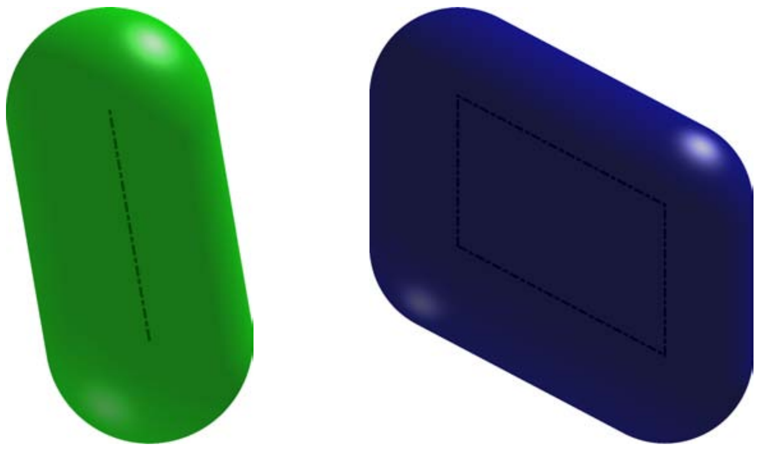
Let's suppose to know the shape of each link of the robot with some parametrization. The next problem, which, as said, is purely geometric, can be solved in different ways.

3.1 Solid interpenetration using Swept Sphere Volumes (SSV)

The first method we provide uses a solid called *Swept Sphere Volume (SSV)*. This method has been used in several works [1] [2].

A SSV is a volume obtained with a sphere moving along a primitive, such as a point, a line segment or a polygon. As a result, its geometric description is very easy and it exploits the rotational invariance of the sphere (fig. 3.1). In fact, to know the displacement of a SSV it is enough to know the sphere radius and only a few points of the primitive, i.e. the point coordinates if the primitive is a point, the extrema

1 a *false negative* happens when a test result indicates that a condition failed, while it actually was successful, in this case it means that no collision was detected while there was a true collision.



(a) Linear SSV **(b)** Polygonal SSV

Figure 3.1: Different types of SSVs.

Linear SSV and Polygonal SSV

points if it is a segment and the vertices of the polygon in the latter case. From now on we refer to *linear SSV* when the primitive is a line segment and to *polygonal SSV* when it is a polygon. The former case with a single point is a degeneration of the linear SSV and practically it is simply a sphere. Between these three, the description which combines the most simplicity (as we see later on, one of them is a little more complex in a collision detection manner) and usability is the linear SSV, since the polygonal SSV can be seen as an extension of the linear one and the sphere is too wasteful of volume.

Drawbacks of SSV modelling

There are a few drawbacks using this description as well. The main one is that the tightness of the shape can be lost if the real joint shape is not well described with a SSV, that is, if it is a parallelepiped, the SSV will include also a volume which doesn't really belong to the joint. To reduce the wasted volume, different strategies can be used:

- use polygonal SSV instead of linear SSV, which can be useful when the joint shape is almost flat;
- use more than one SSV to describe a joint.

To accomplish these methods, a user interface has been created, which lets the user choose how many and which type of SSV to use for each joint. This is described in Section 3.3.

Once the type of SSV is chosen, we shall find a method to test whether two SSVs are interpenetrating or not. In the second alternative described before we will repeat this for each combination of SSVs between the two joints. As a result, there are 3 situations, that is, the SSVs can be both linear, polygonal or one linear and one polygonal. In general, the collision detection phase between two joints using SSV modelling will be referred as *SSV algorithm*.

3.1.1 Interpenetration between two linear SSVs (LL algorithm)

This is the most frequent case, since it is the simplest one. Due to the geometric description of the SSV, we can refer to its primitive. The problem is then reduced to find the distance between two line segments. To do this we apply the algorithm described in [5], which explains the idea behind it and then offers a fast implementation which is also robust to numeric problems. Here we present the theoretical aspect, leaving the reader to the original document for the implementation details.

Distance between line segments

We start describing the line segments in parametric form. The only things we need to describe them are the 4 endpoints, here $\mathbf{P}_1$ and $\mathbf{P}_2$ for the first segment s_1 and $\mathbf{Q}_1$ and $\mathbf{Q}_2$ for the second one s_2, where all the points belong to $\mathbb{R}^3$:

$$\mathbf{P}(s) = \mathbf{P}_1 + s(\mathbf{P}_2 - \mathbf{P}_1) \tag{3.1}$$

$$\mathbf{Q}(s) = \mathbf{Q}_1 + t(\mathbf{Q}_2 - \mathbf{Q}_1) \tag{3.2}$$

where $s \in [0,1]$ and $t \in [0,1]$. As a distance measure we simply use the *euclidean distance*, which from now on will be referred to simply as *distance*. The squared distance between two generic points belonging to different line segments is

$$R(s,t) = \left\|\mathbf{P}(s) - \mathbf{Q}(t)\right\|^2 = as^2 - 2bst + ct^2 + 2ds - 2et + f \tag{3.3}$$

where

$$a = (\mathbf{P}_2 - \mathbf{P}_1)(\mathbf{P}_2 - \mathbf{P}_1) \qquad b = (\mathbf{P}_2 - \mathbf{P}_1)(\mathbf{Q}_2 - \mathbf{Q}_1)$$
$$c = (\mathbf{Q}_2 - \mathbf{Q}_1)(\mathbf{Q}_2 - \mathbf{Q}_1) \qquad d = (\mathbf{P}_2 - \mathbf{P}_1)(\mathbf{P}_1 - \mathbf{Q}_1)$$
$$e = (\mathbf{Q}_2 - \mathbf{Q}_1)(\mathbf{P}_1 - \mathbf{Q}_1) \qquad f = (\mathbf{P}_1 - \mathbf{Q}_1)(\mathbf{P}_1 - \mathbf{Q}_1)$$

The goal is then to find the *minimum* distance between two generic points, that is, we need to find (s,t) which minimizes $R(s,t)$ over the domain $[0,1] \times [0,1]$. Since (3.3) is a continuously differentiable function, the problem is reduced to find the minimum points on the domain boundary and the critical points inside it. The candidates on the boundary are the four corners and four points on the edge, which can be found in [5]. To do the second step, we just need to compute the gradient of (3.3), which is

$$\nabla R(s,t) = 2(as - bt + d, -bs + ct - e) \tag{3.4}$$

and find its zeros, which amount is one if the segments are non-parallel. Conversely, if they are parallel, the minimum distance points are infinite, but their distance is always the same, so we can pick up any pair (s,t) which nullifies (3.4).

A brute force algorithm would compute directly these nine points, but this would be rather slow. Since this is the bottleneck of the entire phase, we must pay attention to each step of the algorithm and try to avoid any unnecessary operation and all heavy computations. All these considerations are met on the algorithm described in [5], which is the one we implemented.

Once we know the distance between two segments $d(s_1, s_2)$, the collision test is straightforward.

Proposition 1 (Collision between linear SSVs)**.** *Let s_1 and s_2 be two line segments parametrized as (3.1) and (3.2), let $d(s_1, s_2)$ be their minimum distance and let r_1 and r_2 be the radii which generate the linear SSVs originated in s_1 and s_2. The SSVs are interpenetrating if and only if*

$$d(s_1, s_2) \leqslant r_1 + r_2 \tag{3.5}$$

Proof. Let's suppose the condition (3.5) is met, then let $\mathbf{P}_0$ and $\mathbf{Q}_0$ be the minimum distance points belonging to s_1 and s_2 respectively. Then the spheres which origins are $\mathbf{P}_0$ and $\mathbf{Q}_0$ intersect because the distance between the origins is less than the sum of the radii, and these spheres are included in the respective SSV by definition.
Conversely, two SSVs interpenetrate only if there exists two spheres (one for each one), whose centres lie on the originating primitive, which intersect. Let $\{(\mathbf{P}_i, \mathbf{Q}_i)\}$ be the set of pair of points for which the SSVs are interpenetrating. For any of these pairs the condition for the relative spheres to intersect is that

$$d(\mathbf{P}_i, \mathbf{Q}_i) = \|\mathbf{P}_i - \mathbf{Q}_i\| \leqslant r_1 + r_2$$

In particular, this condition still holds for the pair $(\mathbf{P}_0, \mathbf{Q}_0)$ for which the distance $\|\mathbf{P}_i - \mathbf{Q}_i\|$ is minimum, but, since

$$\bar{d} = d(\mathbf{P}_0, \mathbf{Q}_0) \leqslant d(\mathbf{P}_i, \mathbf{Q}_i) \leqslant r_1 + r_2$$

holds for any pair in the set mentioned before, this means that (3.5) is met and $\mathbf{P}_0$ and $\mathbf{Q}_0$ are the minimum distance points on the segments s_1 and s_2. □

3.1.2 Interpenetration between linear and polygonal SSV (LP algorithm)

Despite being an extension for the linear SSV method, the collision detection algorithm when at least one SSV is polygonal is not directly descending from the linear case. The main difference is that the volume inside the sweep (which is highlighted in fig. 3.2) is part of the SSV as well. To cope with this, we had to develop an ad-hoc algorithm.

The first thing to notice is that, as we will see later on, the algorithm is reasonably fast if the number of edges of the polygon is small, otherwise we could lose all the advantages that this method is exploiting. Usually we will have to deal with triangular or quadrilateral shapes, so we are not interested in complex polygons.

Another thing to notice is that we are only interested in *convex polygons* because the shapes of the links we are representing are hard to represent with a polygon which isn't convex but has only few edges. There are also some practical reason for this choice, as some steps on the algorithm result in an easier and faster solution.

Before going on, some conventions are imposed:

Conventions about polygonal SSV

- ⋄ for each joint, a SSV reference frame is computed from the Denavit-Hartenberg joint frame. The transformation matrix $\mathbf{T}_{Si}$ is made in 2 steps:
  - the DH frame is rotated so that the Z axis is orthogonal to the plane which contains the SSV polygon;

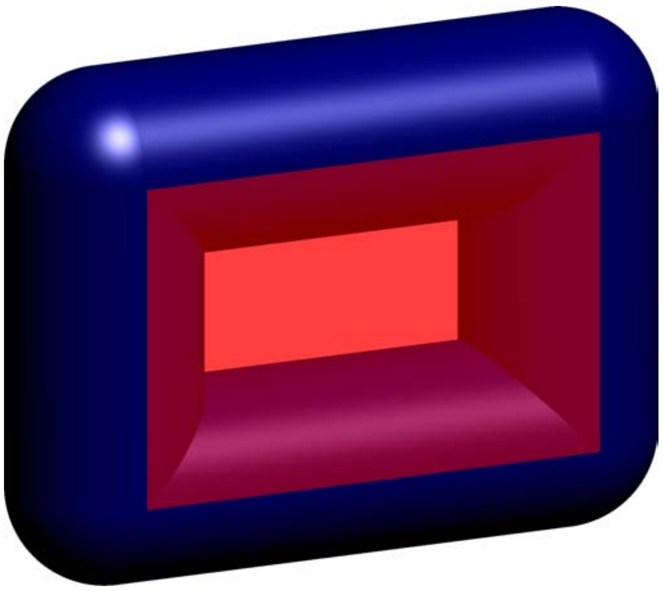


Figure 3.2: Polygonal SSV additional volume. Comparing to the linear case, the volume highlighted in red is part of the SSV as well.

- the rotated frame is shifted along the current Z axis by a value imposed by the user.

◇ all the vertices of the polygon are enumerated in a counter-clockwise sense with respect to the SSV frame.

The first observation lets us reduce the algorithm complexity as for some steps it can be seen as a planar problem. We now summarize the algorithm for collision detection. As for the linear case, the problem can be studied with refer to the primitives of each SSV.

Algorithm 1 Linear-Polygonal SSV collision detection (LP algorithm)

Input arguments:

- endpoints of linear SSV segment
- vertices of polygonal SSV
- radii of each SSV (R_l and R_p)
- transformation matrix between SSVs reference frame

1. Transform linear SSV points into polygonal SSV reference frame
2. Test z coordinate of (transformed) linear SSV endpoints (polygonal SSV's z-level is 0 in this reference frame): if

$$\begin{cases} z_1 > R_p + R_l \\ z_2 > R_p + R_l \end{cases} \quad \text{or} \quad \begin{cases} z_1 < -(R_p + R_l) \\ z_2 < -(R_p + R_l) \end{cases}$$

is satisfied, signal no collision and exit (*Out Of Range shortcut*)

3. If sign(z_1) $\neq$ sign(z_2), go to step A (*intersection with plane*), otherwise go to step B (*no intersection with plane*)

A.1 Find the intersection point $P_0 = (x_0, y_0, 0)$ between the segment of the linear SSV and the plane of the polygonal SSV

A.2 (*Out Of Range shortcut*) Given polygonal SSV points

$$P_i = (x_i, y_i, 0)\ ,\ i = 1, \dots, n$$

(where n is the number of vertices of the polygon), if at least one of these conditions is met

$$\begin{aligned} x_0 &> x_i + R_p + R_l & \forall i = 1, \dots, n \\ y_0 &> y_i + R_p + R_l & \forall i = 1, \dots, n \\ x_0 &< x_i - R_p - R_l & \forall i = 1, \dots, n \\ y_0 &< y_i - R_p - R_l & \forall i = 1, \dots, n \end{aligned}$$

signal no collision and exit.

A.3 If intersection point P_0 is inside the polygon, signal collision and exit

A.4 Apply LL algorithm between the original linear SSV and each linear SSV obtained with all the edges of the polygon. Signal collision if any of these tests returns collision, otherwise there is no collision. In both cases, exit.

B.1 (*Projection Out Of Range shortcut*) Given the points described in step A.2 and the endpoints $P_j = \{(x_j, y_j, z_j)\}_{j=1,2}$ of the linear SSV, if at least one of these conditions is met

$$\begin{aligned} x_j &> x_i + R_p + R_l & \forall i = 1, \dots, n,\ j = 1, 2 \\ y_j &> y_i + R_p + R_l & \forall i = 1, \dots, n,\ j = 1, 2 \\ x_j &< x_i - R_p - R_l & \forall i = 1, \dots, n,\ j = 1, 2 \\ y_j &< y_i - R_p - R_l & \forall i = 1, \dots, n,\ j = 1, 2 \end{aligned}$$

signal no collision and exit.

B.2 Apply LL algorithm between the original linear SSV and each linear SSV obtained with all the edges of the polygon. If the result of any one is collision presence, signal collision and exit.

B.3 For all the endpoints $\mathbf{P}_i$ of the linear SSV for which holds

$$|z_i| < R_p + R_l \tag{3.6}$$

test if their projection onto the polygonal SSV plane is inside the polygon. If this is the case for at least one point, signal collision, otherwise there is no collision. Exit the algorithm.

Sufficient conditions in LP algorithm

There are a few observations to make about Algorithm 1. First thing to notice is that some steps are not really needed since they only give a sufficient condition for no collision. In particular, steps A.2 and B.1 only check whether the point coordinates of linear SSV are out of range respect to the polygon vertices coordinates and could be skipped. Step 2 instead handles the case when each endpoint of the linear SSV is far enough from the plane of the polygon (fig. 3.3a). Though these steps seem to be unnecessary, they are useful because they can avoid all next operations that are computationally heavier, leading to a shorter collision detection time in most of the situations where the SSVs are far from each other.

Coordinates transformation into polygonal SSV reference frame

Step 1 requires a transformation matrix to map the points of linear SSV into the reference frame given by the polygonal SSV. This algorithm is executed once we know the joint configuration of both robots, so the transformation matrix $\mathbf{T}_{is}$ which transforms any point in joint i (Denavit-Hartenberg) frame into station frame is known a priori. The transformation matrix we need transforms the endpoints of linear SSV in DH frame into SSV frame of polygonal SSV. Let $\mathbf{T}_{lp}$ be this matrix. To compute it we need to pass through a common reference frame between the robots, which is the station frame. So this matrix can be split into the product of two matrices which have the station as reference frame:

$$\mathbf{T}_{lp} = \mathbf{T}_{sp}\mathbf{T}_{ls} = \left(\mathbf{T}_{ps}\right)^{-1}\mathbf{T}_{ls} \tag{3.7}$$

$\mathbf{T}_{ls}$ is exactly $\mathbf{T}_{is}$ for the linear SSV, assuming we know SSV points in this reference frame. We need only to find $\mathbf{T}_{ps}$. This matrix can be split as the product of two consecutive transformations, that is

$$\mathbf{T}_{ps} = \mathbf{T}_{is}\mathbf{T}_{pi} \tag{3.8}$$

where this time $\mathbf{T}_{is}$ is referred to the other robot (to avoid any confusion, we will refer to it with a superscript $^{(p)}$). Last transformation matrix is defined by our convention and it is a static matrix which can be calculated offline. This is given by the parameters imposed by user initial setup (see Section 3.3). Combining (3.7) with (3.8) and exploiting the properties of inverse matrices we obtain

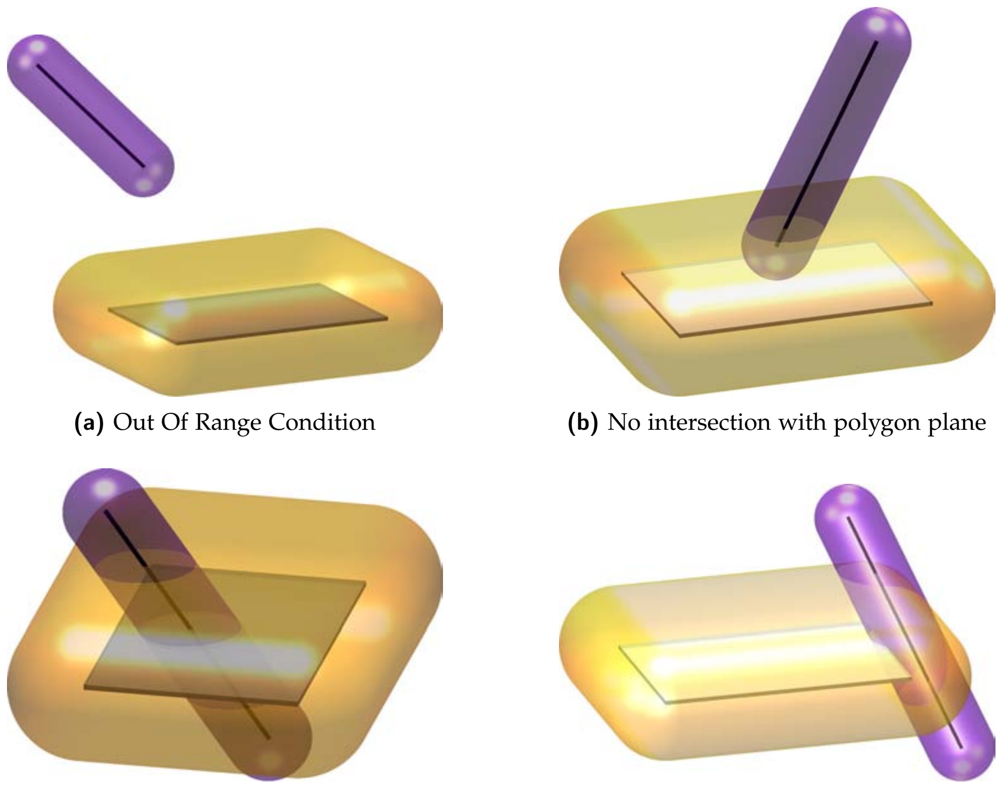
$$\mathbf{T}_{lp} = \mathbf{T}_{pi}^{-1} \cdot \left(\mathbf{T}_{ia}^{(p)}\right)^{-1} \cdot \mathbf{T}_{ia}^{(l)} \tag{3.9}$$

where the inversion can be computed easily exploiting the advantages of transformation matrices (see Appendix B).

Another thing to notice about Algorithm 1 is that the transformation of coordinates allows us to not use the parametrization of the plane, since any problem of finding whether a point is inside a polygon is reduced to a planar problem. A similar problem is solved in Section 3.2, where it remains in 3D, and we will see that a cross product is needed. Though it is a simple operation, it is still more complex than transforming few point coordinates, for which the matrices (i. e.$\mathbf{T}_{is}$) must be computed anyway for other operations on the kinematic model. For the planar problem considered, we used the algorithm explained in [6].

Step A overview

The step A handles the sub-case when there is intersection between the linear SSV segment and the plane containing the polygon. Basically there can be 2 situations:


(a) Out Of Range Condition **(b)** No intersection with polygon plane

(c) Intersection with polygon area **(d)** No intersection with polygon area

Figure 3.3: Different situations in LP algorithm.

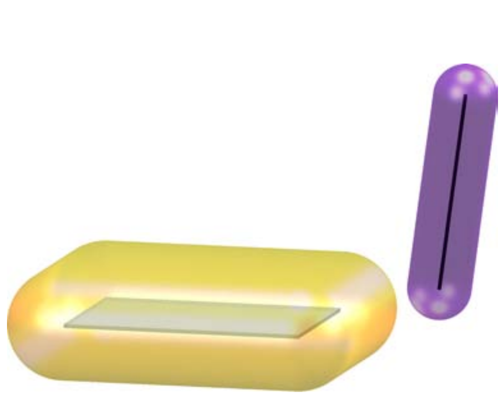
◇ the intersection point is inside the polygon

◇ the intersection point is outside the polygon

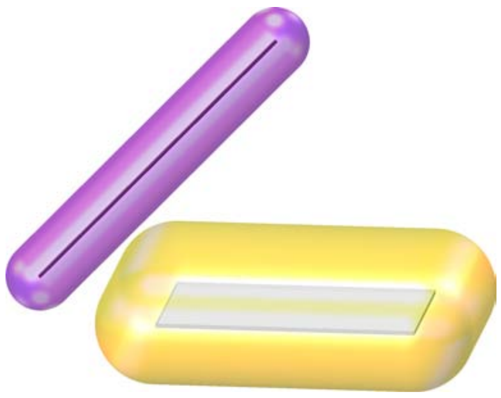
The first case, depicted in fig. 3.3c, leads straightforwardly to a collision presence and it is made in step A.3, while the handling of the latter one (fig. 3.3d) is given to the LL algorithm using all the edges of the polygonal SSV, which returns a necessary and sufficient condition for the collision presence, as the only case which would not be treated would be the one where the linear SSV intersects only the middle volume of the polygonal SSV, which is handled on the step before.

Step B overview

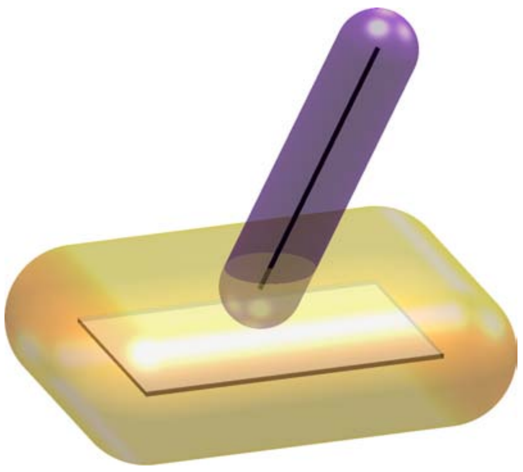
The step B is the sub-case when there is no intersection between the segment and the plane containing the polygon, but the linear SSV is not far enough from the plane. In this case there can be two similar situations, but here they are solved in a different way. Step B.1 exploits the projection of the endpoints of linear SSV into the plane of the polygonal one. Practically this means only to neglect the z coordinate of the endpoints. In this way, situations like the one plotted in fig. 3.4a can lead to determine straightforwardly collision absence because the projections are both outside the range of any polygon vertex. On the other hand, situation in fig. 3.4b will be solved on step B.2 which is handled as step A.4, but this time this gives only a sufficient condition for collision presence. In fact, if there is no collision, the segment could be positioned as in fig. 3.4c. In this case, if the projection of at least one of the endpoints of the linear SSV is inside the polygon, there is collision, while close-to-



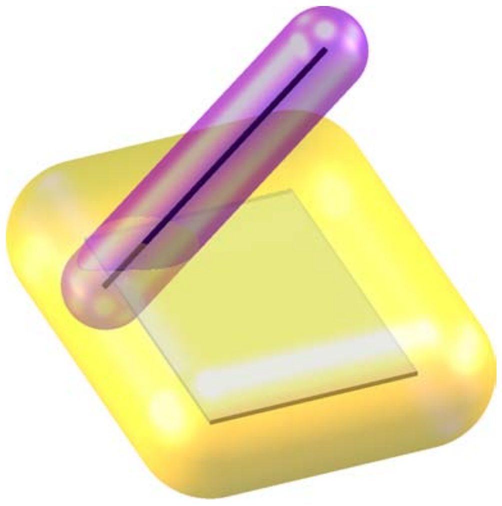
(a) Projection Out Of Range condition



(b) Collision detectable with LL algorithm



(c) Potential collision only with middle volume



(d) Collision correctly handled in step B.2

Figure 3.4: Different situations in LP algorithm in absence of intersection with the polygon plane.

edge conditions (fig. 3.4d) do not generate any trouble since they must have been handled on the step before. This is the reason why this step is performed at last.

3.1.3 Interpenetration between two polygonal SSVs (PP algorithm)

This is the most critical algorithm, as it involves the most complex primitives. What we have done is simply iterating LP algorithm, treating each edge of one polygon as linear SSV and the other one correctly as polygonal. In this way we are sure to detect all possible collisions as there must be an intersection between a linear SSV made from one of the edges and the other polygonal SSV, that is, there cannot be intersection between the central volumes but not between one edge and the central volume.

We did not search for a computationally better solution since in this occasion the shortcuts present in LP algorithm will be usually exploited several times.

3.1.4 Complexity of SSV modelling algorithm

In this section we will explain why this type of modelling is attractive, giving a computational complexity for each algorithm developed before. Since the modelling phase is made directly by the user, it is fundamental to keep the complexity low so we can exploit all the advantages of this technique.

Complexity of LL algorithm

The first algorithm presented was the *LL algorithm*. This is the simplest one since the other two extend this and contain a step with LL algorithm[2]. For each pair of linear SSVs, collision detection is computed in constant time, so the complexity of LL algorithm is $\mathcal{O}(1)$. This is why it is advisable to model as many joints as possible with a linear SSV. If joint j_1 is modelled with p_1 linear SSVs and j_2 with p_2 linear SSVs, LL algorithm is iterated for each combination of joints belonging to opposite robots, that is, this is performed $p_1 p_2$ times, and the collision detection performance is $\mathcal{O}(p_1 p_2)$, which means its complexity grows very fast if we use many SSVs for each joint. This could be justified with a significant better shaping of the joint.

Complexity of LP algorithm

If we look to Algorithm 1, the first steps are performed in constant time, so they can be neglected. The complexity of LP algorithm is then determined by the most complex route between A and B. Let v be the number of vertices of the polygon. Steps $A.1$ and $A.2$ are once again $\mathcal{O}(1)$. The test for P_0 to be inside the polygon in step $A.3$ is instead $\mathcal{O}(v)$ since the complexity is proportional to the number of vertices. Finally, step $A.4$ is LL algorithm iterated v times, then it is again $\mathcal{O}(v)$. On the other hand, step $B.1$ is $\mathcal{O}(1)$, while step $B.2$ (as $A.4$) is $\mathcal{O}(v)$ and step $B.3$ is $\mathcal{O}(v)$ as step $A.3$.
In conclusion the routes have the same complexity and the whole LP algorithm has complexity $\mathcal{O}(v)$. With this in mind, it is advisable to limit the number of edges in the polygon, even though the shortcuts in the average case should improve the performances. In this sense it could be more convenient to model a joint with a polygonal SSV rather than with more linear SSVs, because in that case there is no shortcut and all the collision detections must be performed. By the way, since this involves the *average* case, this cannot be proved, but depends on experimental conditions.

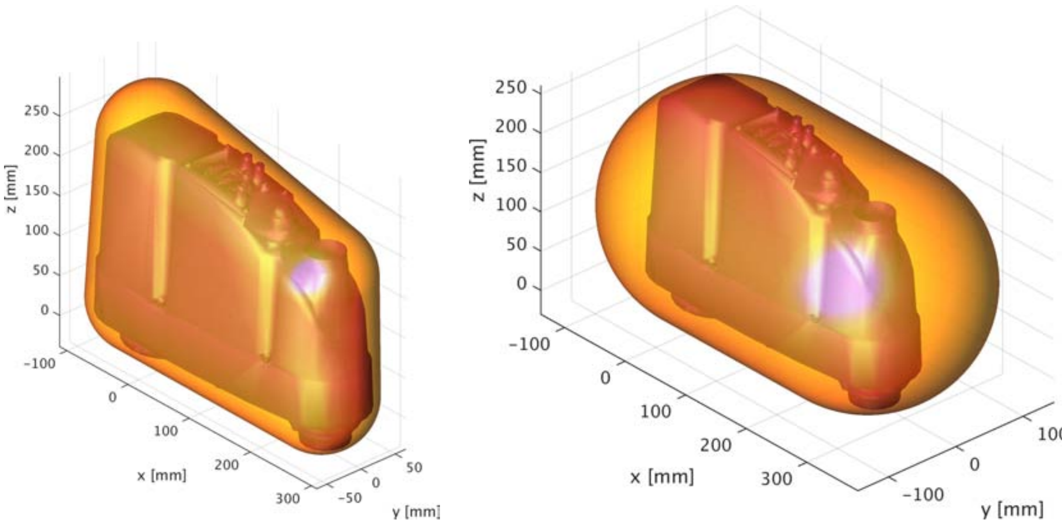
Complexity of PP algorithm

Finally, since *PP Algorithm* is an extension of *LP Algorithm* iterated more times, its complexity is proportional to the number of vertices of each SSV. Let v_1 and v_2 be the number of vertices of each polygonal SSV. PP algorithm iterates LP algorithm for v_1 times, so the complexity is raised to $\mathcal{O}(v_1 v_2)$ and grows very fast if v_1 and v_2 are not small. This is another reason to prefer single linear SSVs to these models, but sometimes the shaping is a lot better with polygonal SSV, like in arm 2 of SCARA Epson® G6653SW (fig. 3.5), where the shape is almost flat. Moreover, this a typical situation if the robots for which we are testing collision presence are identical and we are reasonably using the same model, so the number of joints we are modelling with polygonal SSVs must be as low as possible.

Complexity of collision detection between entire robots

Seeing the entire problem as a collision detection between robots, all these algorithms must be performed for each combination of joint pairs. Let n_1 and n_2 be the number of joints of each robot, let l_1 and l_2 be the number of linear SSVs and let

2 even though there are shortcuts to avoid the steps with LL algorithm, we are interested to the worst case condition when we calculate computational complexity



(a) Quadrilateral SSV modelling **(b)** Single linear SSV modelling

Figure 3.5: Different modelling of arm 2 of Epson® G6653SW (SCARA robot). The linear SSV modelling is obtained with the automatic procedure given by the user interface (see Section 3.3) and it is the tightest volume obtainable with only one linear SSV, but still a lot more wasteful than the polygonal modelling.

p_1 and p_2 be the amount of polygonal SSVs used for the whole kinematic chain. The fastest execution, as mentioned before, is when there is only one linear SSV for each joint, i.e., $l_1 = n_1$, $l_2 = n_2$ and $p_1 = p_2 = 0$. In this case the complexity of collision detection is $\mathcal{O}(n_1 n_2)$ since this is the number of calls to LL algorithm. More generally, collision detection algorithm will consider all pairs of SSVs from each robot, so it can be easily seen that LL algorithm will be invoked $l_1 l_2$ times, LP algorithm for $l_1 p_2 + p_1 l_2$ times and PP algorithm for $p_1 p_2$ times. To consider all these cases we can notice that the whole complexity is proportional to the number of times that distance between two line segments algorithm is called, which is $\sum_i p_i v_i$ on LP algorithm and $\sum_i \sum_j p_i v_i p_j v_j$ on PP algorithm, where p_i the i-th polygon, v_i is the number of vertices of i-th polygon and i and j on the second summation indicate each robot. If we suppose that $v_i = v$ is constant for each polygon, the total complexity can be resumed into

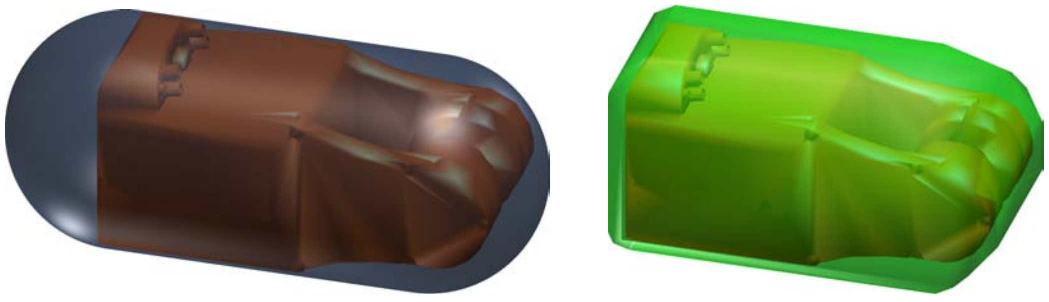
$$\mathcal{O}\Big(l_i l_j + v \sum_i p_i + v^2 \sum_i \sum_j p_i p_j\Big) \tag{3.10}$$

which grows rapidly with the number of polygonal SSVs used.

For example, if we consider two identical anthropomorphous 6-axes robots (so $n_1 = n_2 = 6$) and we model one joint with a quadrilateral SSV and the other ones with single linear SSVs, there are 25 calls to LL, 10 calls to LP and 1 call to PP algorithm and the number of elementary operations are $25 + 40 + 16 = 81$, whereas with only linear SSVs there would be 36 elementary operations (all calls to LL algorithm). Then, in the worst case, just the modelling of one arm with a polygonal SSV could potentially double the time of collision detection, even though practically this is seldom the case.

3.2 Solid interpenetration using boxes

This stage can be seen as an alternative to the previous modelling phase, but more generally, since the results can be different and more accurate, this could be another proper stage in the general collision detection algorithm (cfr. Section 3.4). The main advantage of the SSV modelling is its lightness, while its main drawback is its wasteful of volume. The aim of this modelling is to give a more precise description of each joint (fig. 3.6).



(a) SSV modelling with one linear SSV. **(b)** Box modelling with an 18-DOP box.

Figure 3.6: Different modelling types for the same link (link 4 of 6-axis Epson® C4A601S).

Again, the modelling is done off-line by the user as explained in [10], starting from the CAD model of each link. The primitives we will deal with are convex polyhedra. Before going on, a definition is required.

Definition 1 (Convex polyhedron)**.** *A polyhedron is said* convex *if its surface does not intersect itself and the line segment joining any two points of the polyhedron is contained in the interior or surface.*

From now on, since we are dealing only with these polyhedra, we refer to them simply as *polyhedra*.

Each plane containing each face defines one normal unit vector, which can have 2 opposite directions. We must adopt some conventions, which will be cleared further on:

Polyhedra conventions

- Each face[3] normal points inside the polyhedra (note that this is well defined as the polyhedra is convex)[4];
- All the faces are referred with the labels of each vertex, ordered in counter-clockwise sense seen from inside the polygon.

To satisfy the first condition it is enough to satisfy the second one, then find a point which lies inside the polygon given by the face and then compute the cross product between the vector which goes from this point to any vertex and another vector

3 with face normal we refer to the normal of the plane where the face lies.
4 The important point is that all normals follow the same convention. For example, STL format coming from CAD designs provides the opposite condition which can be easily converted.

which goes from this vertex to the next one. Note that this procedure is always valid because all the faces of a convex polyhedron are convex polygons.

Once these conditions are satisfied, the next proposition holds:

Point inside a polyhedra

Proposition 2. *Let* N *be the number of faces of a convex polyhedron* $\mathcal{P}$, *let* $\{\pi_i\}_{i=1,\dots,N}$ *be the planes containing the faces of the polyhedron parametrized in explicit form so that all the plane normals point inside* $\mathcal{P}$, *i. e.*

$$\pi_i = \left\{(x,y,z)\middle| a_i x + b_i y + c_i z = d_i\right\} \quad \text{where} \quad \mathbf{n} = (a_i, b_i, c_i) \tag{3.11}$$

Then any point $\mathbf{P} = (x,y,z) \in \mathbb{R}^3$ *is inside the polygon if and only if*

$$\begin{cases} a_1 x + b_1 y + c_1 z \geqslant d_1 \\ a_2 x + b_2 y + c_2 z \geqslant d_2 \\ \dots \\ a_N x + b_N y + c_N z \geqslant d_N \end{cases} \tag{3.12}$$

is satisfied by $\mathbf{P}$, *that is, if the point* $\mathbf{P}$ *lies in the* positive half-space *of all the planes.*

It can be proven that the previous proposition is equivalent to the definition of convex polyhedra. To perform collision detection we must test whether two polyhedra are interpentrating.

Interpenetration between convex polyhedra

Definition 2 (Interpenetration between convex polyhedra). *Let* $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *be two convex polyhedra defined with the set of planes* $\left\{\pi_i^{(1)}\right\}_{i=1,\dots,N_1}$ *and* $\left\{\pi_j^{(2)}\right\}_{j=1,\dots,N_2}$ *relative to each face as defined in Proposition 2, where*

$$\pi_i^{(1)} = \left\{(x,y,z)\middle| a_{i1} x + b_{i1} y + c_{i1} z = d_{i1}\right\}$$
$$\pi_j^{(2)} = \left\{(x,y,z)\middle| a_{j2} x + b_{j2} y + c_{j2} z = d_{j2}\right\}$$

The two polyhedra are said to be interpenetrating if there exist a point $\mathbf{P}_0 = (x,y,z) \in \mathbb{R}^3$ *for which*

$$\begin{cases} a_{11} x + b_{11} y + c_{11} z \geqslant d_{11} \\ \dots \\ a_{N_1 1} x + b_{N_1 1} y + c_{N_1 1} z \geqslant d_{N_1 1} \end{cases} \tag{3.13}$$

$$\begin{cases} a_{12} x + b_{12} y + c_{12} z \geqslant d_{12} \\ \dots \\ a_{N_2 2} x + b_{N_2 2} y + c_{N_2 2} z \geqslant d_{N_2 2} \end{cases} \tag{3.14}$$

are both satisfied, if there is another point $\mathbf{P}_1$ *which satisfies (3.13) but not (3.14) and if there is a third point* $\mathbf{P}_2$ *which satisfies (3.14) but not (3.13).*

The previous definition without considering points $\mathbf{P}_1$ and $\mathbf{P}_2$ would include the case where one polyhedra is completely inside the other one, which isn't interesting in our problem since there will be always the crossing between two joints. This is not hard to handle but it leads to unnecessary redundancy in the algorithm that we will present later on.

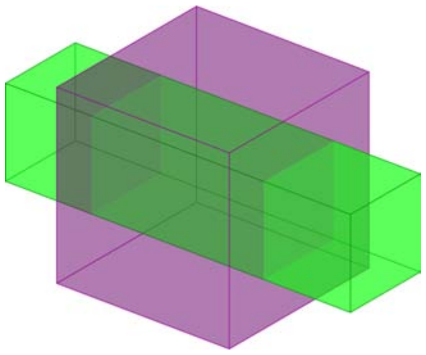
Definition 2 is not practically easy to implement, so we need at least an equivalent condition for two polyhedra to interpenetrate.

Proposition 3. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two convex polyhedra for which one of them is not contained inside the other one. $\mathcal{P}_1$ and $\mathcal{P}_2$ interpenetrate if and only if there is at least one intersection between an edge of one of them and a face of the other one.*

Equivalent condition for interpenetration between convex polyhedra

Proof (informal). The sufficient condition is trivial, since the point of intersection belongs to both polyhedra and then satisfies both (3.13) and (3.14).
Conversely, let us suppose there is interpenetration. If we demonstrate that there exist two faces f_1 and f_2 respectively from $\mathcal{P}_1$ and $\mathcal{P}_2$ which intersect each other, then one of the edges of f_1 or f_2 must intersect the other face. But since the set of points belonging to each polyhedron is a compact set and the faces can be seen as the boundary of this set, this must happen since, for hypothesis, none of these is a subset of the other one and so the boundaries of these two sets must intersect at least in one point. □

Notice that the previous statement requires only that the intersection is from the edge of one polyhedron to the faces of the other one, but it is not necessary that this relationship is reversed. This means that a double pass would be needed to ensure to detect the collision, where we pick all the edges from $\mathcal{P}_1$ and all the faces from $\mathcal{P}_2$ and then we swap the rules. A condition where a single pass would not be enough is depicted in fig. 3.7a.


(a) Suppose we are only checking edges of the purple polyhedron with faces of the green one. In this case the collision would not be found, whereas after swapping the rules it would be detected.

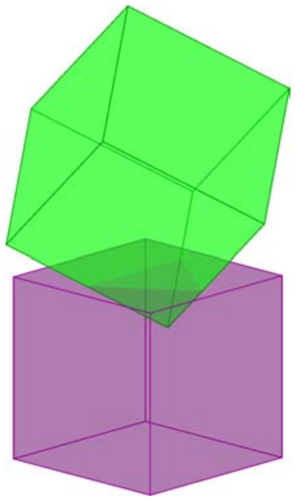

(b) Suppose $\mathcal{P}_1$ is the purple one and $\mathcal{P}_2$ is the green one. If we apply Proposition 4 as presented, collision would not be detected, but it will be after we swap the polyhedra and apply it again.

Figure 3.7: Potential undetected collision.

Proposition 3 can directly be implemented to test collisions between joints, but this algorithm would be quite complex, as we see further on. We need then some sufficient conditions which lead to faster computation in case of collision.

Sufficient condition for collision

Proposition 4. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two polyhedra as in Proposition 2 and let*

$$V_1 = \{v_i\}_{i=1,\dots,N_1} \qquad \textit{where} \qquad v_i = (x_i, y_i, z_i) \in \mathbb{R}^3$$

be the set of vertices of $\mathcal{P}_1$. If $\mathcal{P}_1$ is not completely inside $\mathcal{P}_2$ and if there exists at least one element of V_1 which is inside P_2, that is, if any v_i satisfies (3.14), there is interpenetration between $\mathcal{P}_1$ and $\mathcal{P}_2$.

Proof. It is trivial. Any vertex of $\mathcal{P}_1$ satisfies (3.13) by definition, then if any v_i satisfies (3.14) too the condition in Proposition 2 is met since $\mathcal{P}_1$ is not contained in $\mathcal{P}_2$. ☐

In the previous statement the rules of the polyhedra can obviously be swapped, but it still remains a sufficient condition even with a double pass, so we can handle situations like in fig. 3.7b but not the one depicted in fig. 3.7a.

Since this check is faster than the previous one, it is a good idea to split the collision algorithm in two stages performed serially, as showed in Algorithms 2 and 3.

Algorithm 2 Box collision detection - Stage 1 (*Box1*)

```
Input arguments:
  • set of vertices $V_1$ of polyhedron $\mathcal{P}_1$
  • set of faces $F_2$ of polyhedron $\mathcal{P}_2$

  1. For each face $f_i$ of $F_2$ compute the plane $\pi_i$ which contains it in explicit form
  2. for all $v_i \in V_1$, test if $v_i$ is a solution of (3.14). If at least one is a solution,
     trigger collision and exit algorithm.
```

Algorithm 3 Box collision detection - Stage 2 (*Box2*)

```
Input arguments:
  • set of vertices $V_2$ of polyhedron $\mathcal{P}_2$
  • set of faces $F_1$ and $F_2$ of polyhedra $\mathcal{P}_1$ and $\mathcal{P}_2$

  1. For each face $f_i$ of $F_1$ compute the plane $\pi_i$ which contains it in explicit form
  2. For each face plane $\pi_i$ of $\mathcal{P}_1$
       For each face $f_j$ in $F_2$:
         For each edge in $f_j$:
           2.A parametrize edge, which endpoints are $\mathbf{P}_1 = (x_1, y_1, z_1)$ and
               $\mathbf{P}_2 = (x_2, y_2, z_2)$:
```

$$\begin{cases} x(t) = x_1 + (1-t)(x_2 - x_1) \\ y(t) = y_1 + (1-t)(y_2 - y_1) \\ z(t) = z_1 + (1-t)(z_2 - z_1) \end{cases} \tag{3.15}$$

2.B find if current edge is parallel to the current face plane π_i, i.e., if $\pi_i = \{(x, y, z) | a_i x + b_i y + c_i z = d_i\}$ test if

$$a_i(x_2 - x_1) + b_i(y_2 - y_1) + c_i(z_2 - z_1) = 0 \tag{3.16}$$

If (3.16) is true, skip to the next edge

2.C find the intersection point between the line that generates the current edge and the plane π_i of polyhedra $\mathcal{P}_1$:

$$\begin{cases} x(t) = x_1 + (1-t)(x_2 - x_1) \\ y(t) = y_1 + (1-t)(y_2 - y_1) \\ z(t) = z_1 + (1-t)(z_2 - z_1) \\ a_i x(t) + b_i y(t) + c_i z(t) = d_i \end{cases} \tag{3.17}$$

which gives

$$\tilde{t} = -\frac{a_i x_1 + b_i y_1 + c_i z_1 - d}{a_i(x_2 - x_1) + b_i(y_2 - y_1) + c_i(z_2 - z_1)} \tag{3.18}$$

2.D if $\tilde{t} \notin [0, 1]$, skip to next edge, otherwise build intersection point between the edge and the plane $\tilde{\mathbf{P}} = (x(\tilde{t}), y(\tilde{t}), z(\tilde{t}))$.

2.E *(Out Of Range Shortcut)* Check if any of the coordinates of $\tilde{\mathbf{P}}$ is out of the range of the same coordinate of all vertices of the face f_i of polyhedra $\mathcal{P}_1$, i.e. check if at least one of the following is satisfied:

$$\begin{aligned} x(\tilde{t}) &> x_j & \forall j &= 1, \ldots, n_i \\ x(\tilde{t}) &< x_j & \forall j &= 1, \ldots, n_i \\ y(\tilde{t}) &> y_j & \forall j &= 1, \ldots, n_i \\ y(\tilde{t}) &< y_j & \forall j &= 1, \ldots, n_i \\ z(\tilde{t}) &> z_j & \forall j &= 1, \ldots, n_i \\ z(\tilde{t}) &< z_j & \forall j &= 1, \ldots, n_i \end{aligned}$$

where n_i is the number of edges of face f_i. If so, move to next edge

2.F Check whether $\tilde{\mathbf{P}}$ is inside face f_i. If so, signal collision, otherwise move to next edge

3. If no collision has been detected, swap polyhedron $\mathcal{P}_1$ with $\mathcal{P}_2$ and perform steps 1 and 2
4. If still no collision is detected, trigger no collision and exit.

Algorithm Box2 requires to check whether a point is inside a polygon in 3D space. Despite being a planar problem, it is handled as a non-reduced problem for a simple reason: to reduce it to a 2D problem we would need the transformation matrix of the plane, which is not directly available. To compute it we would need to evaluate a cross product and then a coordinate transformation, which is a little bit more complex compared to what we developed in Algorithm 4 (cfr. fig. 3.8).

Algorithm 4 Point inside polygon in 3D space

Input arguments:

- set of vertices $V = \mathbf{v}_1, \dots, \mathbf{v}_n$ of polygon in 3D space
- normal $\mathbf{n}$ of the plane containing the polygon
- target point $\mathbf{P}$

Output argument: boolean which is true if point is inside polygon

```
for $i = 1$ to $n$ do
    $\mathbf{v}_{12} = \mathbf{v}_{i+1} - \mathbf{v}_i$
    $\mathbf{v}_{2p} = \mathbf{v}_{i+1} - \mathbf{P}$
    $\mathbf{v}_n = \mathbf{v}_{12} \times \mathbf{v}_{2p}$
    if $verse(\mathbf{v}_n) \neq verse(\mathbf{n})$ then
        return false
    end if
end for
return true
```

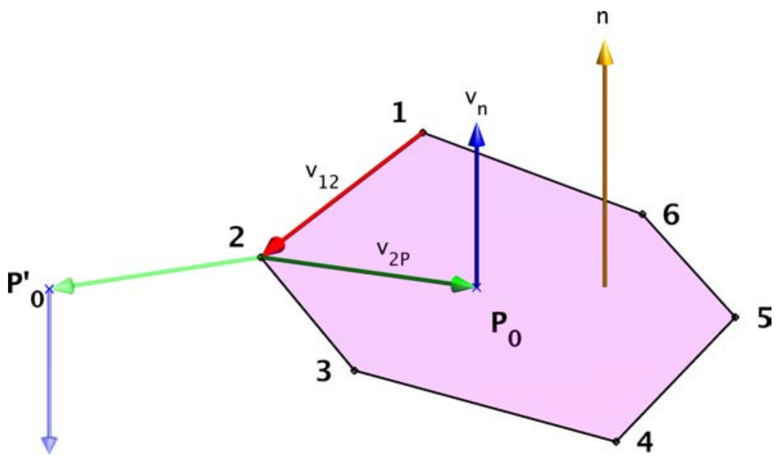


Figure 3.8: Vectors considered to test if a point is inside a polygon.

Notice that $\mathbf{v}_n$ and $\mathbf{n}$ are always parallel unless $\mathbf{P}$ is on the perimeter of the polygon, in which case instead of testing the equiverse condition we simply denote as inside it. Moreover, vector $\mathbf{v}_{i+1}$ will be $\mathbf{v}_1$ when $i = n$. To test the equiverse condition we can simply test if

$$\|\mathbf{v}_n + \mathbf{n}\| > \|\mathbf{v}_n\| + \|\mathbf{n}\| \tag{3.19}$$

Observations about Box Algorithms

As can be seen, stage 1 of the algorithm is just the sufficient condition expressed in Proposition 4 and so it does not guarantee to reach the goal. As said, before continuing to stage 2 we can swap the rules of the polyhedra and re-apply stage 1. Conversely, Box2 algorithm is just an implementation of Proposition 3. It is composed by 3 loops which iterate through all the faces of $\mathcal{P}_1$, then all the faces of $\mathcal{P}_2$ and finally all edges of each face in $\mathcal{P}_2$. In the inner loop it checks whether the current edge of $\mathcal{P}_2$ is crossing the current face of $\mathcal{P}_1$. This results in a quite heavy computation, and this is why this overall algorithm is not as fast as the ones presented with SSV method.

Another observation is that step 2.E can be skipped, but in practice this allows to skip test 2.F most of the times, leading to a serious improvement in practical situations. Point 2.F is needed since until step 2.D we only know that the edge is crossing the plane containing the other face, but we don't know if it crosses it inside the area of the other face.

Conditions for Box algorithm speed-up

Moreover, it must be said that the situation depicted in fig. 3.7a is quite unusual since there must be no vertex of the purple box which crosses any face the green one. Usually the crossing is going to happen at least one frame before. If we neglect this situation we can have a final outcome of collision detection just performing Box1 with a single pass and Box2 skipping step 3, paying attention to the roles of $\mathcal{P}_1$ and $\mathcal{P}_2$, which must be the same through the two stages (i. e. they have to be as described in the outline). In fact, even with the double pass, the situation depicted in fig. 3.9a cannot be handled with Box1, while collision can be detected with one single pass of Box2. On the other hand, if $\mathcal{P}_2$ is the purple polyhedron in fig. 3.9b and we apply

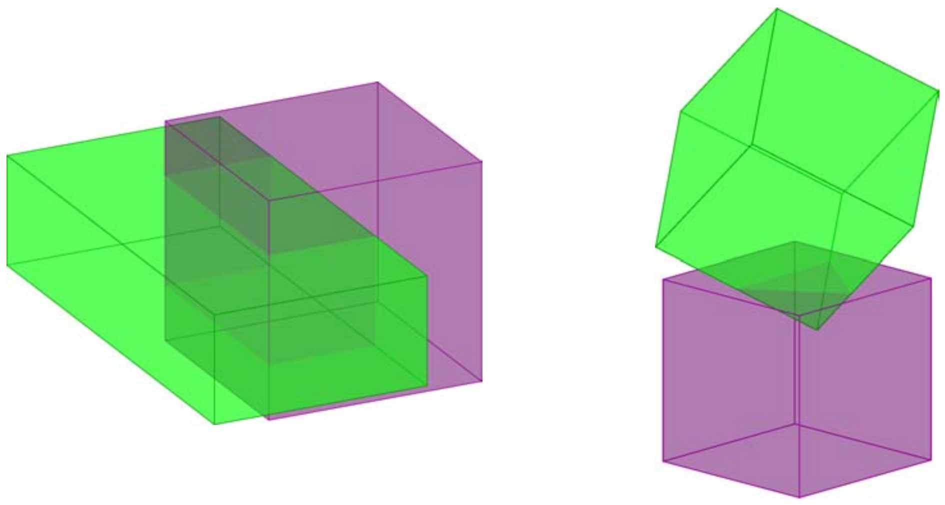

(a) Box1 algorithm unhandled situation **(b)** Single pass Box2 algorithm potentially unhandled situation

Figure 3.9: Box1 and Box2 algorithms potentially undetected collisions.

Box2 without step 3, we wouldn't detect the collision, but this would be done by previous Box1 where $\mathcal{P}_2$ will still be the purple one since there is a vertex of $\mathcal{P}_1$ inside $\mathcal{P}_2$. In this way we can avoid the second pass of Box2 algorithm, speeding up the collision detection computation. We will refer to the series *Box1-Box2* (without step 3) simply as *Box Algorithm*.

3.2.1 Complexity of collision detection algorithm using boxes

If we want to quantify the comparison between this procedure and the SSV method, we must calculate the computational complexity of each algorithm employed.

Complexity of Box1 algorithm

First of all, let's consider *Box1* algorithm. As we can see in Algorithm 2, the two operations are made serially, so the cost is given by the most complex of them. Step 1 operates only on two vertices of each face so its complexity is[5] $\mathcal{O}(|F_2|)$. On the other hand, step 2 has to evaluate a system of $|F_2|$ inequalities for $|V_1|$ points, i. e., its complexity is $\mathcal{O}(|V_1||F_2|)$ and it is dominant to the previous value. In conclusion, the complexity of *Box1* algorithm is $\mathcal{O}(|V_1||F_2|)$.

Complexity of Box2 algorithm

Let's now have a look at *Box2* algorithm. We are not interested in step 1 since the others will be far more complex. As explained before, there are three nested loops, then the inner loop is computed[6] $|F_1||F_2|e_2$ times, where e_2 is the mean number of edges of each face in $\mathcal{P}_2$, which is usually constant. The inner loop's main cost is given by the function which tests if a point is inside a polygon, which is Algorithm 4. As this needs to iterate through all vertices of the polygon, its complexity is $\mathcal{O}(e_1)$. Then, the total complexity of *Box2* algorithm is $\mathcal{O}(|F_1||F_2|e_1e_2)$, which, if the order of magnitude of the number of faces and edges of the polyhedra is the same (let's say f and e), this can be resumed into $\mathcal{O}(f^2e^2)$. If we consider the general robot collision detection phase with n_1 and n_2 arms, total complexity is simply $\mathcal{O}(n_1n_2f^2e^2)$.

Comparison with SSV algorithms

The first observation is that, as expected, *Box1* algorithm is a lot faster to perform. The other thing is that, since we are interested in the worst case, the total complexity of this method is given by the heaviest one, which is *Box2*. If we compare this with the result obtained in Section 3.1.4, even though the quantities are not properly comparable, we see that this is far more complex than the SSV modelling method. To state this, it is easy to see it with an example: let's suppose both boxes are parallelepipeds, so $f = 6$, $e = 4$ and $v = 8$. First stage leads to 48 operations, while second stage consists of almost 576 elementary operations. Even just considering the order of magnitude of these numbers, it easy to see that the SSV modelling is a lot faster. Moreover the modelling with simple parallelepipeds doesn't justify the use of this method since its aim is to obtain a more precise representation of each joint, which is done using more complex polyhedra, as explained in [10], for which the number of edges and faces is larger than this simple example.

It is easy to see that the worst case happens when there is no collision, while in some situations when there is collision the detection could be found even at first stage. As we will see on Section 3.4, if we are able to invoke *Box* algorithm only when we are almost sure there is a collision, we can combine the fastness of SSV modelling and the precision of box modelling together to get a reliable solution for collision detection.

5 with $|\bullet|$ we consider the cardinality of a set in this environment
6 on the worst case, which in this occasion is particularly important since the whole algorithm goes empty when there is no collision, even though the shortcut should be helpful

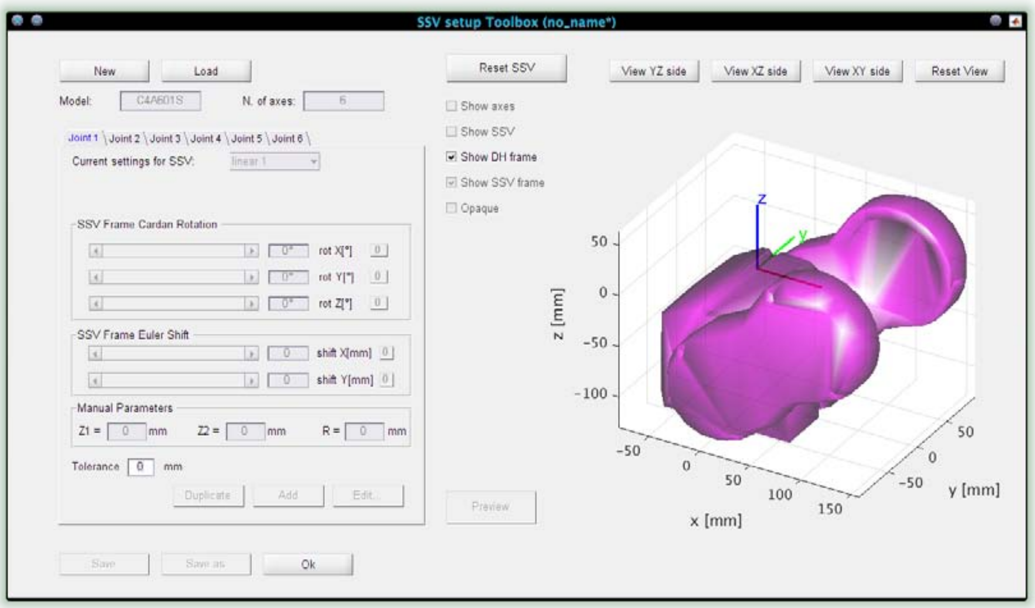
3.3 User interface for initial SSV setup

In this section we present a user interface which aims to design a suitable SSV model for each robot joint. The reason why we did this is because an automatic procedure is not always the best solution. A first problem it may encounter is the possibility to choose between one or more linear SSVs per joint or even polygonal joints, and a choice criteria is not easy to define.

Another problem is that it is not easy to define an optimal SSV shape, as there can be any different criteria to do this. For example, if we consider only a simple linear SSV, the sweep axis can be not coincident with the joint revolution axis, as long as it depends on the joint shape.

Moreover, one may think that the optimal SSV is the one that minimizes its volume, or better which minimizes the ratio between SSV and arm volume. This however might not be a good criteria, as sometimes we can allow a larger volume for a single arm which overlaps with part of the following or previous one. Since then there are a lot of parameters which depends on practical environment, we left the user some degrees of freedom.

The first screen provided to the user is depicted in fig. 3.10.


SSV setup Toolbox (no_name*)

Model: C4A601S
N. of axes: 6
Current settings for SSV: linear 1
rot X[°] 0°
rot Y[°] 0°
rot Z[°] 0°
shift X[mm] 0
shift Y[mm] 0
Z1 = 0 mm
Z2 = 0 mm
R = 0 mm
Tolerance 0 mm
☐ Show axes
☐ Show SSV
☑ Show DH frame
☑ Show SSV frame
☐ Opaque

Figure 3.10: User Interface initial screen.

At this point no SSV is yet built, as it can be seen. Main buttons are:

◇ *New*: it allows to create an initial default SSV configuration. If a model is already loaded, it overwrites all edits;

◇ *Load*: loads a previously saved SSV configuration file;

◇ *Reset SSV*: it restores all edits to their initial values;

◇ *View* buttons: allow to see joint and SSVs from a different view and eventually to reset the initial view. The plot can also be rotated with the mouse when pointing over the plot;

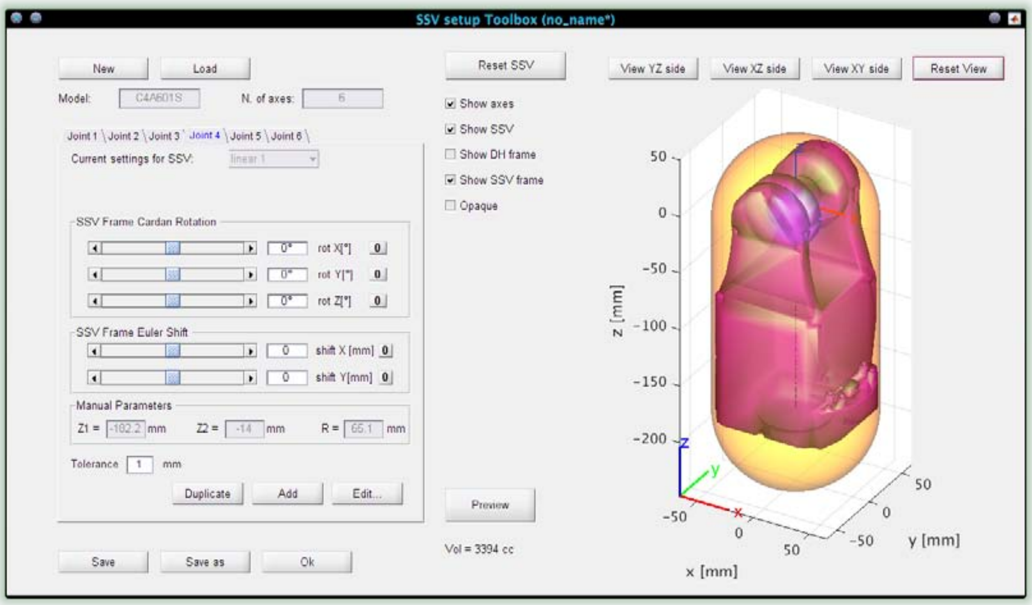
◇ *Show* checkboxes: they allow to show or hide main frames and SSVs;

◇ *Opaque* mode: with this option flagged, the surfaces become opaque. This is useful in manual mode to check whether each joint is covered by the SSVs;

◇ *Preview*: it opens a preview window which shows the entire robot with SSV representations of each joint;

◇ *Save* buttons: they allow to save the new or edited model, offering the option to save with another name (with `.ssv` extension). *Ok* button closes the window and saves any modified settings.

The main section where all edits are performed is split into tabs. Each tab contains the settings for correspondent joint. When no model is loaded, only the tolerance value can be set, which can be different for each joint. In this way the new model that will be created is directly designed with this tolerance value. This represents the tolerance of the SSV radius which is fed into the SSV automatic generation algorithm (see Section 3.3.1). Once a model is loaded or a new model is created, these tabs become editable (fig. 3.11).


SSV setup Toolbox (no_name*)

Model: C4A601S N. of axes: 6

Current settings for SSV: linear 1

SSV Frame Cardan Rotation: 0° rot X[°] 0; 0° rot Y[°] 0; 0° rot Z[°] 0

SSV Frame Euler Shift: 0 shift X [mm] 0; 0 shift Y[mm] 0

Manual Parameters: Z1 = -182.2 mm Z2 = -14 mm R = 65.1 mm

Tolerance 1 mm

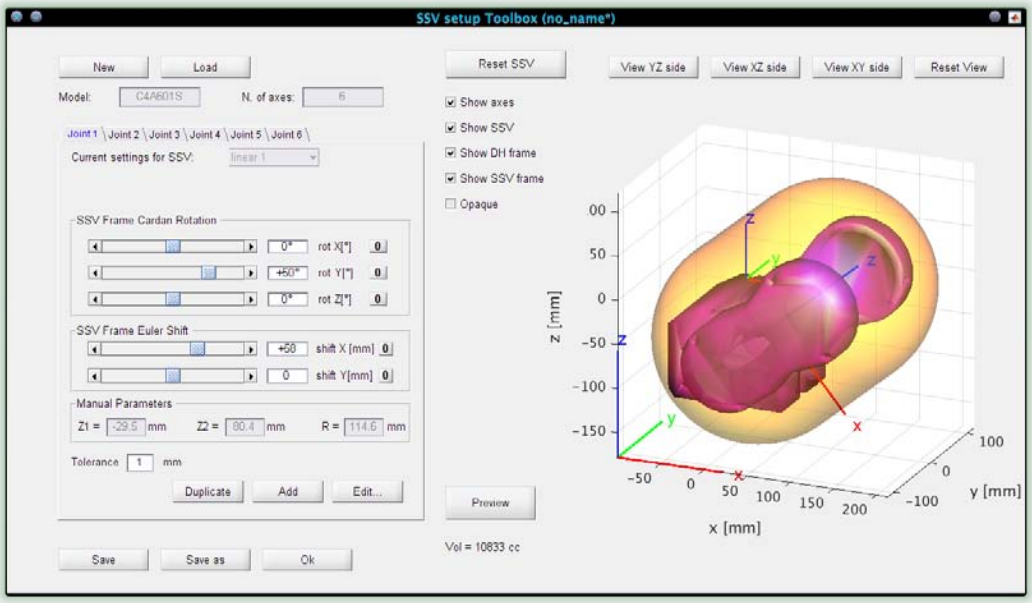
☑ Show axes
☑ Show SSV
☐ Show DH frame
☑ Show SSV frame
☐ Opaque

Vol = 3394 cc

Figure 3.11: User Interface after a new model is loaded.

Automatic mode overview

For each tab, different option may be available according to the SSV type and quantity for the current joint selected. If there is only one simple linear SSV the pop-up menu is not selectable. We refer to this situation as *automatic mode*. If a new model is created, SSV automatic generation procedure is invoked with default parameters. This is explained in Section 3.3.1. In fact the only options available (fig. 3.12) are:



Model: C4A601S N. of axes: 6

Current settings for SSV: linear 1

☑ Show axes
☑ Show SSV
☑ Show DH frame
☑ Show SSV frame
☐ Opaque

SSV Frame Cardan Rotation
0° rot X[°] 0
+50° rot Y[°] 0
0° rot Z[°] 0

SSV Frame Euler Shift
+50 shift X [mm] 0
0 shift Y[mm] 0

Manual Parameters
Z1 = -29.5 mm Z2 = 80.4 mm R = 114.6 mm

Tolerance 1 mm

Vol = 10833 cc

Figure 3.12: User Interface with edited parameters for a linear SSV in automatic mode. SSV frame is the one which is rotated, while other two are DH frame (the one on the corner is simply shifted).

- ◇ *SSV Frame Cardan rotation*: with the 3 sliders or with the edit boxes it is possible to define 3 Cardan angles from which the rotation matrix between the SSV reference frame and the current link DH frame is built;
- ◇ *SSV Frame Euler shift*: with the 2 sliders or with the edit boxes it is possible to define the SSV eccentricity respect to the axis obtained after previous rotation. Since sweep is performed along z axis in SSV frame, it is only necessary to define shifts along local x and y axis.

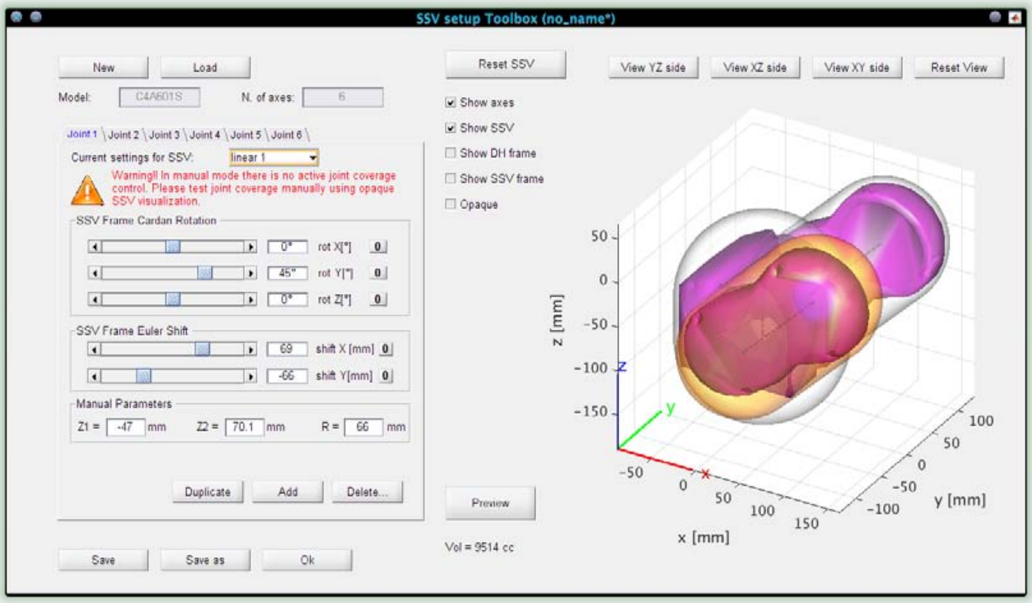
The SSV reference frame is a custom defined frame which endpoints of SSV refer to, in order to simplify following computation. Hence it is static compared to the link DH frame. If we are dealing with linear SSVs, by our convention the z-axis is the one along which the sphere is swept.

In this mode the SSV is continuously updated according to parameters set by the user. The way how this is performed is explained again in Section 3.3.1, where, instead of default values, user parameters are employed. Notice that in this mode user cannot set the radius as it is evaluated automatically. Tolerance value instead can still be edited. Despite not necessarily being the best criteria, SSV volume is also shown in order to compare results while varying parameters.

The bottom push buttons in each tab modify the type or the number of SSVs. In particular, *Edit* button has the purpose of switching the type of SSV between linear and polygonal, *Duplicate* creates an exact copy of the actual SSV, while *Add* simply adds one new linear SSV with default parameters. Once we add a new SSV for the current joint, we switch to *manual mode* (fig. 3.13).

Multi-SSV design manual mode overview

First thing to say is that in this mode there is no automatic procedure to check whether the SSVs cover the whole joint, as recalled in main tab (fig. 3.13). This has



Model: C4A601S N. of axes: 6

Current settings for SSV: linear 1

Warning!! In manual mode there is no active joint coverage control. Please test joint coverage manually using opaque SSV visualization.

SSV Frame Cardan Rotation
0° rot X[°] 0
45° rot Y[°] 0
0° rot Z[°] 0

SSV Frame Euler Shift
69 shift X [mm] 0
-66 shift Y[mm] 0

Manual Parameters
Z1 = -47 mm Z2 = 70.1 mm R = 66 mm

[✓] Show axes
[✓] Show SSV
[ ] Show DH frame
[ ] Show SSV frame
[ ] Opaque

Vol = 9514 cc

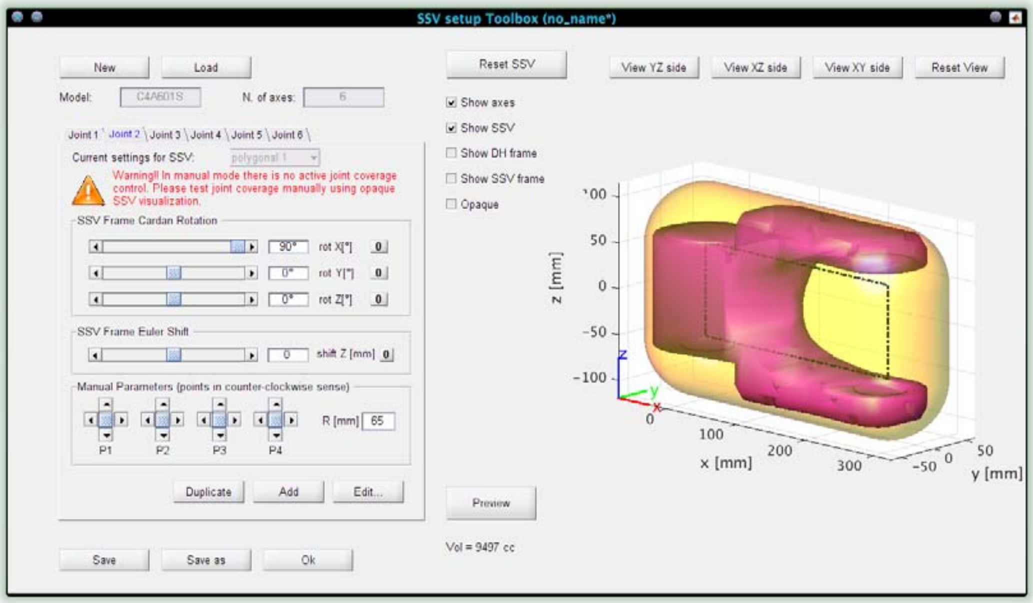
Figure 3.13: User Interface for manual mode, when multiple linear SSVs are selected for each joint. New parameters became available, while selected SSV (for which specified parameters hold) is highlighted on the plot.

been a design choice, since in some way the user would be asked to specify how to split the coverage, which would mean for example to define a plane which splits the joint shape into different parts. Instead of this, we gave the user the possibility to specify the endpoints of each SSV. Clearly, these points are referred to each SSV reference frame, which is specified as before with two transformation matrices. Therefore it is only necessary to specify z quotes of these points. In this occasion the radius of each SSV must be set manually, as there is not any automatic procedure to calculate it. For this reason, the tolerance value has been dropped.

Once entered in manual mode with multiple joints the *Edit* button switches to *Delete* function. In fact, as long as it is possible to add new SSVs it is possible to remove them. This opens another menu which allows to specify which SSV to remove. It is possible to remove SSVs as long as just one linear SSV remains. Notice that removing any SSV but last one leads to leave part of the joint uncovered. If, instead, we are removing the last SSV available (i. e., we are switching from 2 to 1 linear SSV), we switch back to automatic mode. In this way manual parameters are removed and are replaced by automatic ones, while rotation and shift values remain the ones of the last SSV remaining. At this point *Delete* button is replaced again with *Edit* button.

Polygonal SSV design manual mode overview

If we switch to a polygonal SSV, there are new options available (fig. 3.14). We decided to design only quadrilateral SSVs, despite the algorithm works for any convex polygon. Since we are dealing with a planar polygon, we decided to build another reference system, such that the polygon is defined on the xy plane. As for the linear case, this frame is static compared to DH frame. The transformation matrix between these two frames is again computed in two steps, of which the former one is the


SSV setup Toolbox (no_name*)

Model: C4A601S N. of axes: 6

Current settings for SSV: polygonal 1

Warning!! In manual mode there is no active joint coverage control. Please test joint coverage manually using opaque SSV visualization.

SSV Frame Cardan Rotation

90° rot X[°]

0° rot Y[°]

0° rot Z[°]

SSV Frame Euler Shift

0 shift Z [mm]

Manual Parameters (points in counter-clockwise sense)

R [mm] 65

☑ Show axes

☑ Show SSV

☐ Show DH frame

☐ Show SSV frame

☐ Opaque

Vol = 9497 cc

Figure 3.14: User Interface for manual mode, when polygonal SSV modelling is selected for the specific joint.

same as before and it defines rotation parameters, while the offset is defined only with the z quote of the local frame, as we are dealing with a plane.

Once the plane is defined, the user can manually set the four points of the quadrilateral using the sliders in *Manual Parameters* section, baring in mind the final polygon must be convex. The four points are defined in counter-clockwise order referring to the xy plane. As for the multiple case, the user must define the radius, while once again the tolerance value is no more needed.

As it seems, there is no possibility to have both linear and polygonal SSVs for the same joint. This is another design choice. In fact, since both polygonal SSVs and multiple SSVs modelling yet lead to high computational cost themselves, it would be too expensive to combine these strategies for the same joint. As a consequence, if we want to switch from multiple SSVs modelling to polygonal modelling it is necessary to remove all linear SSVs.

3.3.1 SSV automatic generation procedure

This procedure is invoked when we are running in automatic mode. If a new SSV model is created, we set the initial parameters to be all 0 (i. e., SSV frame is coincident with DH frame). The aim of this phase is to create the SSV that has the smallest volume and contains the joint, according to specified rotation and translation. First thing to do is then to convert robot link points into SSV frame, which allows to reduce to a 2D problem, as we will see. In fact all edges we are dealing with are straight, so if the point cloud is contained into the SSV, the robot arm will be contained there as well.

First of all, we have to build transformation matrix $\mathbf{T}_{Si}$ which transforms points from SSV frame to DH frame. This is the inverse of the matrix we are searching for, but it is straightforward to evaluate. Let α, β and γ be the Cardan angles around respectively axes x, y and z imposed by the user. The rotation matrix around fixed reference frame (i. e., DH frame) is

$$\mathbf{R}_{xyz} = \mathbf{R}_z(\gamma)\mathbf{R}_y(\beta)\mathbf{R}_x(\alpha) \qquad \in \mathbb{R}^{3\times 3}$$

For more details about rotation matrices see Appendix B.

Now let x_0 and y_0 be the coordinates which define eccentricity, set up by the user. The transformation matrix which considers both rotation and shift is finally:

$$\mathbf{T}_{Si} = \left[\begin{array}{ccc|c} & & & x_0 \\ & \mathbf{R}_{xyz} & & y_0 \\ & & & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{3.20}$$

If we want to get joint points into SSV frame we must use the inverse of $\mathbf{T}_{Si}$, which is easy to compute (see Appendix B).

This procedure is valid for whatever set of points expressed in DH coordinates, so we can use either points from joint CAD design or vertices from 18-DOP or 30-DOP models. In the former case the resulting SSV is tighter to the joint, whereas in the latter one we can inherit any tolerance imposed from the Bounding Volume computation [10]. To transform points we simply use their homogeneous coordinates and we pre-multiply them by the transformation matrix.

Next step is to compute the radius r of the SSV. According to the current parameters, the z axis in SSV frame is well defined. Hence, for each link point, we need only to evaluate its (minimum) distance from this axis. The radius will be the maximum between all distances found. Since we converted all points into SSV frame, z axis has null x and y coordinates and so we can neglect z coordinate of all points as the problem is radial. For the moment, let's not consider the tolerance, which will be added at the end of this procedure.

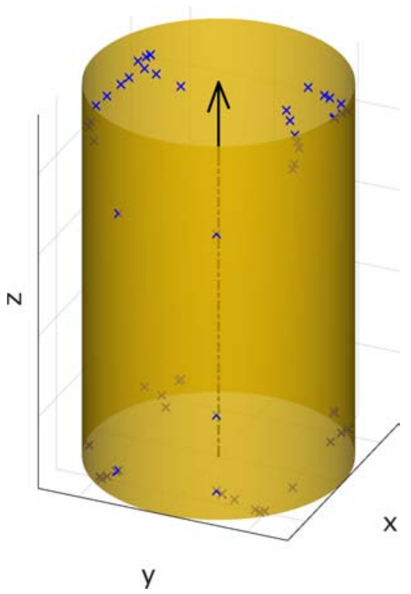
At this point, as we can see in fig. 3.15a, we obtained a cylinder which contains all points. Following step is to find endpoints of SSV primitive. Since this procedure is pretty similar for both top and bottom of SSV (where with *top* we mean that it refers to the endpoint with greater z quote), we will report the steps only for the bottom side.

Firstly, we evaluate the point which has the minimum z coordinate (z_{min}). Then we extract only the subset of points (x_i, y_i, z_i) for which

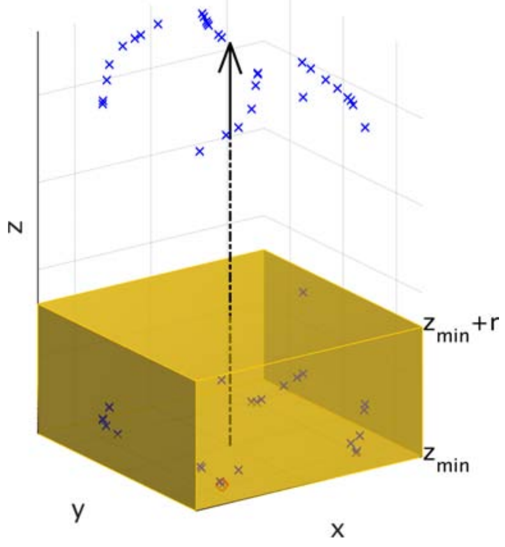
$$z_i \leqslant z_{min} + r$$

We are now searching for the half sphere which contains all points in the volume (fig. 3.15b) defined as:

$$\mathcal{S} = \left\{(x, y, z) : |x| \leqslant r,\ |y| \leqslant r,\ z_{min} \leqslant z \leqslant z_{min} + r\right\} \tag{3.21}$$



(a) Cylinder containing all link points, after having computed its radius.



(b) Subset of points wherewith we compute bottom endpoint. Point which has the lowest z quote is highlighted in red.

Figure 3.15: SSV computation according to point cloud.

The only unknown in this problem is the sphere center z quote (z_c). Whatever is this point, the following relationship must hold:

$$x_i^2 + y_i^2 + (z_i - z_c)^2 \leqslant r^2 \qquad \forall i : z_i \leqslant z_c \tag{3.22}$$

where i subscript refers to the points in $\mathcal{S}$. The previous one is the distance condition from the endpoint, which defines a sphere indeed, of which we are interested only in the bottom half one. Since condition (3.22) must hold as equality for at least one point, we can first of all solve the correspondent equation in z_c for all points in $\mathcal{S}$, and then keep only the maximum among the two solutions. In fact the half sphere we are searching for is pointing toward the bottom. Finally, z_c is the minimum amongst this resulting set, as we have to keep the most restrictive condition.

After having repeated this method for the top endpoint, we increase the radius by the tolerance set up by the user, such that every link point is far at least that amount from the SSV boundary.

3.4 Collision detection general algorithm

In this section we try to combine both types of joint modelling in order to get a solution which is suitable to the user requirements. We will propose different solutions, which can differ for execution time and degree of precision due to particular practical set-up. First of all, we are going to solve two different problems, which can be seen as two different level of abstraction:

◇ Given two joint configurations $\mathbf{q}_1$ and $\mathbf{q}_2$, find if there is collision between the robots;

◇ Given two joint paths $\mathbf{q}_1(s)$ and $\mathbf{q}_2(t)$, find if there is collision over the time.

Most of the first part has been done, but now we are going to combine the two methods proposed.

3.4.1 Collision between two static joint configurations

A explained more times, the main collision detection problem, as we formulated it, is when there is no collision at all. In fact both method proposed rely its worst case in this occasion. With this in mind, it is essential that first of all collision-free situations are handled as fast as possible, and this can be done by SSV algorithms. We can see the purpose of that modelling as not to find real collisions, but as to discard all further checks. In fact, both SSV and Box algorithms give necessary and sufficient conditions for the collision, that is, they can work independently; the main thing for both is that the modelled joints are colliding, but not necessarily the real ones. In this sense, since box modelling can be more accurate, it can approach with more probability the real collision. By the way, the user can yet act here if the only thing that matters is execution time and a raw result is acceptable.

Box algorithm as an improvement of SSV algorithm

What we are trying to do in this phase is to use Box algorithm as an improvement of SSV algorithm to be invoked only when this last one triggers a potential collision. If there is a real collision, the cost given by Box algorithm will be very low. Conversely, every time there is a *false alarm* (i. e., SSV algorithm signals collision but Box does not), the added time given by Box1 and Box2 computation is a loss of performance. The probability of *false alarm* relies all on the accuracy of SSV modelling. Therefore the user shall solve this trade-off: the more accurate is SSV modelling, the more time will be spent on SSV algorithm, but the less is the probability to trigger a false alarm and waste time on Box algorithm. This depends on practical factors: if the robots are very close to each other, SSV modelling should be as tight as possible, whereas, if not, the probability of false alarm could be already low and a raw first stage modelling could be enough.

Another thing to notice is that, if there are n_1 and n_2 joints, ideally it would be necessary to check $n_1 n_2$ collisions between pair of joints. Depending on real robot displacement, some of these checks might be pointless as the corresponding links might not be physically touching, regardless of the joint configuration. This is typical for the first arm of two anthropomorphous robots (fig. 3.16). As a consequence, we provide the user with a simple interface which allows him to exclude collision check between any pair of joints. In a future improvement, this can easily be done automatically.

Priority checks

Extending this concept, some pair of joints might be colliding with more probability than other ones. Without any constraint, we are going to perform collision detection serially checking a joint of one robot with all the other ones of the other robot. If we rule the checking phase with a priority system, the time needed to find a collision might be even reduced, without any drawback. The checking order can be set by the user and depends on both relative robot position and trajectory and it is not simple to describe nor it is going to work for all possible trajectories, but

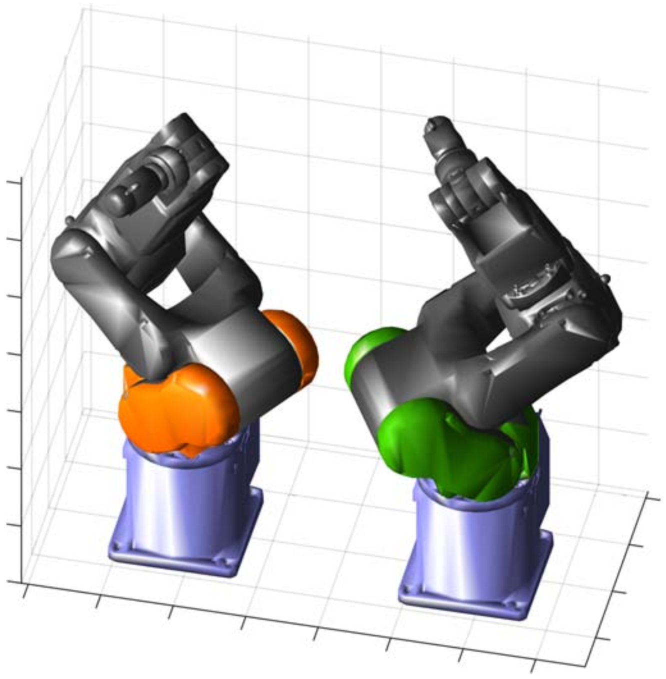


Figure 3.16: A workcell example with two Epson® C4A601S anthropomorphous robots, where each arm 1 (the one highlighted) cannot phisically touch the other arm 1 since the bases are fixed to ground.

can always perform the less probable checks by last. Moreover, it is improving the performance only in case of collision, but it does not affect at all the performance time when there is no collision.

Box modelling sub-levels

Box modelling can always be split in different sub-levels, as a simple parallelepiped (OBB) reduces the computation time compared to a more complex box, such an 18-DOP or a 30-DOP. In this way any false alarm given by SSV algorithm can be captured with a simpler model, which reduces the waste of time thanks to the skipped computation of Box2 algorithm with a complex model. However, if Box algorithm with an OBB model triggers a collision, another call to Box algorithm with the most precise model is required to get the final outcome, unless the precision of OBB model is enough. In this sense, the intermediate level could lead to another waste of time or it could speed up the algorithm, depending on the environment. For this reason, we developed more than one possible way to get to the final result.

3.4.1.1 *Collision detection hybrid algorithms*

In this section we refer to *level* when we want to indicate the level of modelling precision:

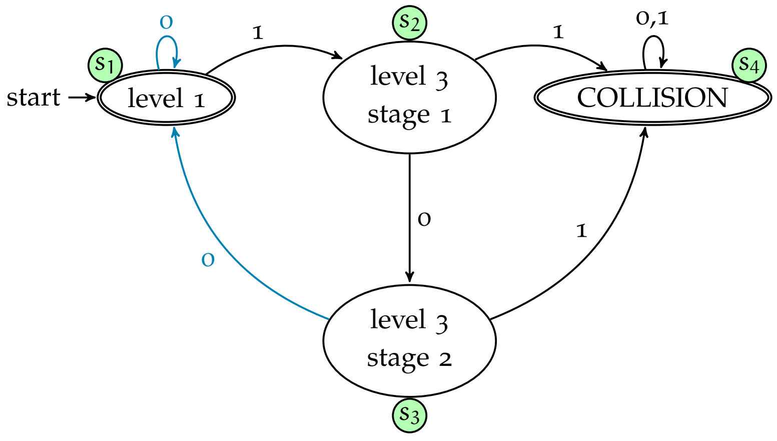
- level 1 is the SSV modelling;
- level 2 is the Box modelling with OBB;
- level 3 is the Box modelling with 18-DOP or 30-DOP[7].

7 Depending on requirements the user can choose between these two, baring in mind that a 30-DOP is even more complex than a 18-DOP

When we refer to the pair *level/stage* we are considering either Box1 or Box2 algorithm, depending on the *stage* value. The outline of the algorithm is at joint level of abstraction, that is, we are testing the collision for a certain pair of joints until we have a definitive outcome, and then we repeat the same for the next pair.

The easiest way to explain the hybrid algorithm is using a *Deterministic Finite Automaton*, in fact we can see each stage of the algorithm as its states. One final state of this automaton consists of a state which is reached whenever a collision is finally detected. For example, we can impose that a collision can be recognised as detected only when we are performing an algorithm at level 3, which is the most accurate in our case. Any other collision-detected trigger signal just moves the computation to the next level. To compact the notation, we indicate the initial state as a final state as well, baring in mind that we can exit the algorithm only when there is no collision between the entire robots, that is, only when there is no collision between the last of the pair of joints we are checking. If this happens for another pair of joints but last, the transition is highlighted and we start again from first state with the next pair of joints in the check-list.

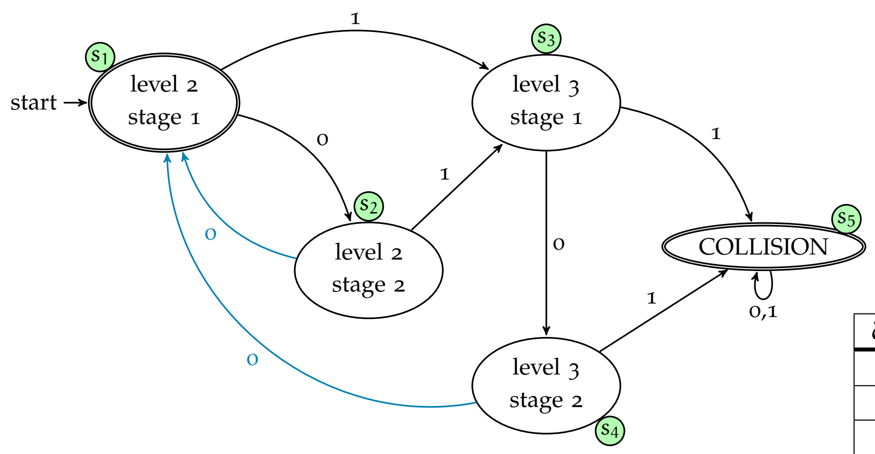
AUTOMATON 1: LEVEL 1-3 The first automaton we propose skips level 2 and moves directly to level 3 as soon as a collision is detected at level 1 (fig. 3.17). The reason why this is a good choice is explained before, since the second step can be a wasteful of time. The choice of this DFA must result from practical considerations: if we know a priori that the probability of false alarm is low, we are almost sure that when level 1 triggers a collision it is a true collision and then the intermediate level wouldn't be helpful. This, as said, is useful when the robots are not very close to each other or the SSV modelling is quite tight. In all automata considered here transitions labelled with '0' mean that on the previous state no collision was detected, while '1' means that a collision was detected.



$\delta(s,\bullet)$	**0**	**1**
s_1	s_1	s_2
s_2	s_3	s_4
s_3	s_1	s_4
s_4	s_4	s_4

Figure 3.17: Automaton 1: Level 1-3. The blue transition allows to proceed with following pair of joints. On the right, transition function which represents it.

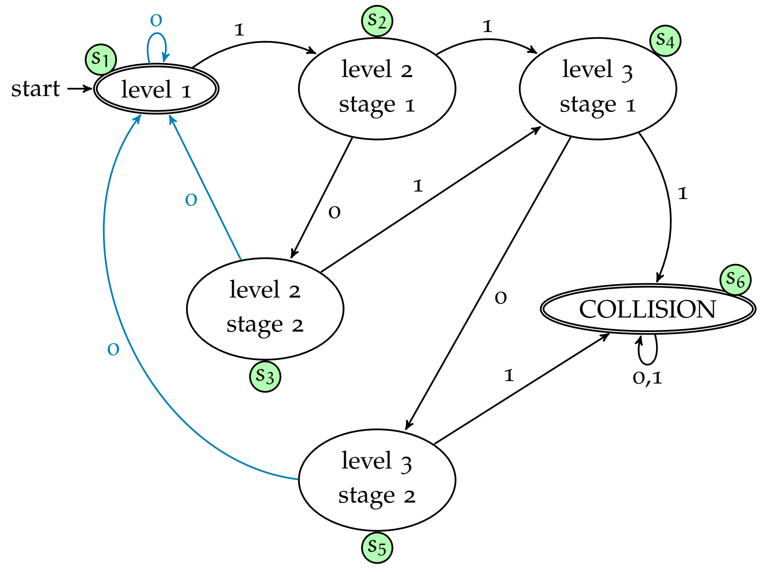
AUTOMATON 2: LEVEL 2-3 The second automaton we present does not exploit all the advantages of SSV modelling (fig. 3.18). For this reason, it is not going to be used practically, but since it is very easy to implement a new automaton (we only need to store the transition table) we developed it just for comparison reasons, in order to understand the benefits given by SSV modelling.



$\delta(s,\bullet)$	0	1
s_1	s_2	s_3
s_2	s_1	s_3
s_3	s_4	s_5
s_4	s_1	s_5
s_5	s_5	s_5

Figure 3.18: Automaton 2: Stage 2-3. On the right, transition function which represents it.

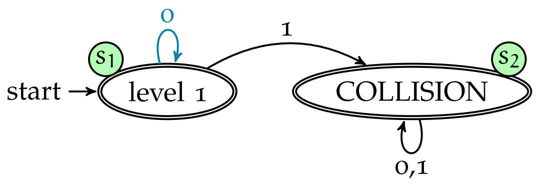
AUTOMATON 3: LEVEL 1-2-3 This automaton can be seen as the complete one and it performs all three levels (fig. 3.19). As said for the first automaton, the second level could represent an improvement depending on practical situation.



$\delta(s,\bullet)$	0	1
s_1	s_1	s_2
s_2	s_3	s_4
s_3	s_0	s_4
s_4	s_5	s_6
s_5	s_0	s_6
s_6	s_6	s_6

Figure 3.19: Automaton 3: Level 1-2-3. On the right, transition function which represents it.

AUTOMATON 4: LEVEL 1 This automaton is the fastest one and exploits only SSV modelling (fig. 3.20). In this sense it can be seen as the complementary of Automaton 2. As a consequence it inherits all the pros and cons of SSV modelling. By the way, depending on the specifications, it can always be a valid alternative.



$\delta(s,\bullet)$	**0**	**1**
$\mathbf{s_1}$	s_1	s_2
$\mathbf{s_2}$	s_2	s_2

Figure 3.20: Automaton 4: Level 1. On the right, transition function which represents it.

Automata variations A variation of these automata is to perform stages 1 with the double pass, trying to prevent to invoke *Box2* algorithm when there is a true collision. This can be useful when the probability that a vertex of one polyhedron is inside the other polyhedron is high, which is common with level 3 when the number of vertices is large.

Another alternative could be to perform Box algorithm with different models, for example with one joint represented with an OBB an the other one as a 18-DOP. By the way, there could be benefits only in case of real collision, but since, after some tests, we found that the main cost of the entire algorithm is given by collision-free configuration pairs, this has not been implemented because the benefit would be negligible.

3.4.2 Collision detection between trajectories

We now move one abstraction level forward. Given two joint configurations $\mathbf{q}_1(\bar{t})$ and $\mathbf{q}_2(\bar{u})$, we can now know if there is collision invoking the hybrid algorithm explained above. We will assume that now the configurations can change through time, so we have the sets

$$\begin{aligned} \mathcal{Q}_1 &= \{\mathbf{q}_1(t),\ t = 1,\dots,N_1 \in \mathbb{N}\} \\ \mathcal{Q}_2 &= \{\mathbf{q}_2(u),\ u = 1,\dots,N_2 \in \mathbb{N}\} \end{aligned}$$

where t and u are just discrete times and do not coincide with real time displacement, and N_1 and N_2 are the number of configurations each trajectory has along time.

For the next procedure we can omit the real time distribution, that is, the mapping between the discrete time t (and u) and the real time when the particular configuration is reached. This is done because we can not know the time dependency but only a chronological sequence of positions, as in our case. However, we can have additional informations which can improve the procedure, as we will see.

If we don't have any information about the real time nor we have the correlation between t and u, the only thing we can do is to check the collision for every possible pair

$$\big(\mathbf{q}_1(t), \mathbf{q}_2(u)\big) \quad \forall (t,u) \in [1, N_1] \times [1, N_2]$$

A typical situation is when one of the robots (let's say robot 1 with trajectory $\mathbf{q}_1(t)$) has started its movement and the other one is steady and waiting to start its one. In this condition, we refer to *Master Robot* as the moving robot and to *Slave Robot* to the one which is querying about collision with the master one. Since we could not even know at which time the slave is querying for collision, we must check all its temporal sequence with every $t \in [1, N_1]$, even though the master current time is N_1.

The first thing we could do is simply to loop collision detection for each time sequence:

```
for $t = 1$ to $N_1$ do
    for $u = 1$ to $N_2$ do
        testCollision($\mathbf{q}_1(t), \mathbf{q}_2(u)$)
        if collision then
            return
        end if
    end for
end for
```

Alternative check sequence

This simple procedure guarantees to get to the final result, but if the collision happens at the end of the trajectory its computation time is quite long. We can improve computation time when there is collision exploiting correlation between path points. Even though we don't know the exact time distribution above the sequence of points, in the optimal check sequence we would check the pair of joint configuration for which its collision probability conditioned by past collision-free time pairs is the maximum among all the remaining pairs.
Let $C_{t,u}$ be the event "collision between robots with configuration $\mathbf{q}_1(t)$ and $\mathbf{q}_2(u)$" and let us indicate with $\bar{C}$ its complementary. Let's suppose we didn't check none of the future pairs $(\bar{t}, \bar{u})$, with $\bar{t} > t$ and $\bar{u} > u$. The previous affirmation means that usually it doesn't make sense to check pair $(t, u+1)$ if the previous check was at (t, u) since

$$P(C_{t,u+1} | \bar{C}_{t,u}) \leqslant P(C_{t,u+\alpha} | \bar{C}_{t,u}), \quad \alpha > 1 \tag{3.23}$$

This is because the prediction error variance on the right side of (3.23) tends to increase with the rise of α as the correlation between the collision events through far moments tends to zero. On the other hand, if there is a collision, it can be found for a different time pair as well, thanks to the correlation between adjacent moments and to the fact that any collision usually involves more moments.

As said, this should only give an idea on how to proceed, since theoretically all these aspects should be taken in a more strictly way. We can't follow a formal way since we don't know the time distribution and because it wouldn't be easy to model the collision probability between two robot configurations.

We tried to exploit this with another checking sequence. The idea is to scan all the possible pairs in more stages, where the first stage has to cover only the most significant pairs and the last one has to do it with all the remaining ones. We set as most significant pairs the ones that correspond to a via point pass-through, which are the first to be checked. If a collision happens in between two via points, it might be handled on the first stage as well, as usually the collision lasts for at least a few moments. We followed the next 3 stages:

- On the first pass we check the collision between only the initial and the final moments, i. e., pairs $(1,1),(N_1,N_2),(N_2,1),(1,N_2)$;
- On the second stage we check all via points pairs[8];
- Finally we scan all remaining pairs if still no collision is detected.

Smart time sequence check

A smart choice for the last step is to cycle through all (remaining) times of slave robot in the inner cycle and leave the outer cycle for the master ones, which will be performed in the opposite sense (i. e., from the last moments to the first ones of the master robot trajectory). In this way, supposing t_N is the time correspondent to the last via point of the Master Robot, when the inner cycle is completed and no collision is found, we are sure that there is no collision between the Slave Robot and the last leg of the MR trajectory. This is useful if during this phase we get the information that the MR has passed through the last via point, so we can start the movement of the Slave one, regardless of the remaining collision test outcome, since those potential collisions belong to the past. This obviously can be extended to all via points of the Master Robot. In fact we are considering the worst case condition, that is, we might be querying for collision as soon as MR has begun its movement, so we shall check collision with the whole trajectory, but in practice there can be various situations.

Priority pairs alternative

Another alternative is to split the collision algorithm in two parts at joint level: by first we perform collision detection only between a subset of joint pairs, called *prior pairs*, and then we do the same for all the others. So we repeat the entire sequence through time only for prior pairs, then we start again and we do the same for the remaining joint pairs. This can be useful when we know that collision is frequent between some particular pair of joints, such as any one which which includes the end-effector. In this way if there is collision the time required to reveal it can be even shortened. However no benefit is given if there is no collision at all or if the collision involves non-prior joints. In this case, instead, there is a performance loss as each robot kinematic model must be computed twice, as we will scan twice all possible time pairs. A solution could be to store the kinematic model for each time if possible, but eventually the benefits of this method must be tested on the field.

8 if CP is on, there is no passing through via points, but, as stated in Chapter 2, we can use the point which is in the middle of the junction which avoids the via point instead

4

Algorithm Optimization

Despite the improvement obtained using hybrid algorithms, the algorithm developed in Chapter 3 cannot achieve real-time performance on most situations when there is no collision. Real-time constraints superimposed require all the collision detection stage to be performed in less than 0.1 s. This can be achieved only in case of real collision, but in many critical collision-free paths the results obtained are not satisfactory.

In particular, one situation which is not acceptable to be slowly detected is when the robots are very far or when they are working in safety conditions, for example in opposite areas (fig. 4.1).

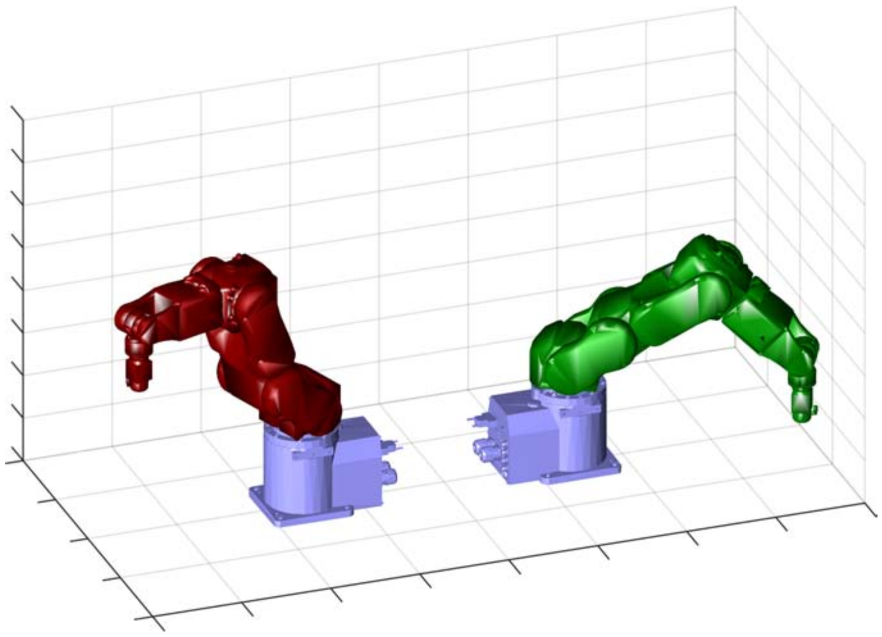


Figure 4.1: Potentially safe working condition between two robots sharing the same workspace.

In this case there is clearly no collision but all steps of the algorithm must be performed anyway. This particular situation let us think another method which would be adopted prior to the collision detection algorithm at its actual state.

At this stage we had to focus only on anthropomorphous manipulators, to which the methods offered are applicable. We considered this case because it is the most common and the most interesting, as well as the most complex. If one or more robots are SCARA, this method can still be applied with slightly variations, but its efficiency should be verified. By the way, in this occasion there could be even more efficient methods, based on the limited motion a SCARA robot can perform.

To perform this optimization we have to move one step backward to what we reached at the end of Chapter 3, as long as this is going to replace the last optimizations explained in Section 3.4.2 with more efficient solutions.

4.1 Robot Maximum Swept Volume approach

At this point our algorithm does not exploit any information about the joint configuration (it only uses the kinematic model, but not explicitly their value). The next step is then based on the sweep of the robot volume across the time sequence. This method works well only when both the robots lie on parallel planes and the z direction of the robot frame is the same. In other words the results might be poor if one of the two robots is wall-mounted or ceiling-mounted, even though the idea can be extended to these cases.

The idea behind the next part is that most information about the obstruction of an anthropomorphous robot with first rotational axis orthogonal to the working plane (as it is on most anthropomorphous robots) is given by the first two joint positions θ_1 and θ_2 (fig. 4.2). The approach we followed is to first represent the robot shape

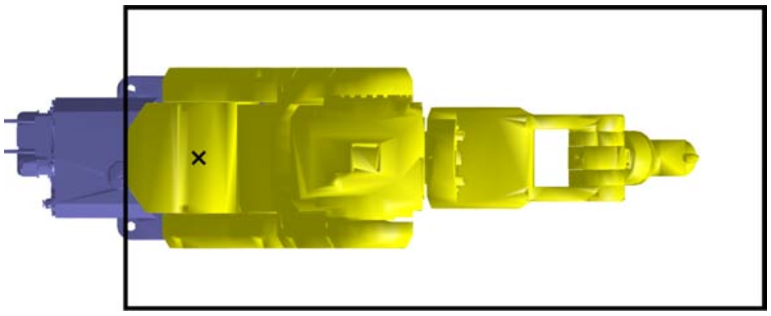


Figure 4.2: Top view of a robot configuration and its bounding box obtained considering first two joints as fixed and all possible combinations of the other four.

with a simple volume and then sweep it rotating it according to the first joint position (fig. 4.4). We must bare in mind that this phase must be a sufficient condition for collision absence, so we have to consider all possible configurations which can happen along the trajectory. In Section 4.1.1 we develop a way to compute the robot obstruction given a certain joint configuration, while its transformation into a proper volume is described in Section 4.1.2.

4.1.1 Box obstruction according to first joint position

The basic idea which all these methods must follow is that this stage must be performed prior to the algorithm developed by now, so their critical properties must be its fastness and simpleness. As a result, we chose to reduce this problem as a planar problem, as it is seen from a top view (fig. 4.2). In this way we neglect the z coordinate, that is, we treat the bounding box as a rectangular parallelepiped[1]. On

1 from now on we refer to it simply as parallelepiped

the other hand this is a simplification and the bounding box could be wasteful, but if the robots are very far from each other we can achieve good results. Conversely, if one of the robot is wall or ceiling mounted z axis neglection could lead to poor results as the sweep volumes may overlap frequently without any collision. This is the reason why this specific method is not suitable in this situations.

Bounding Box computation with joint 1 locked

The next step is then to find the rectangle which contains the plan view of the robot, given a specific set of configurations. To do this, we imagine to lock the first joint and then to move all other joints across all possible values they assume in the set, so we are sure that, once we compute the swept volume (Section 4.1.2), the robot is contained within the volume obtained.

This rectangle could be computed on-line according to the current trajectory, but it wouldn't be as fast as required, and certainly a brute force implementation cannot reach the goal in time. We decided instead to do another approximation which allows to make an off-line implementation of this method. First of all we must point out that the width of the rectangle is mainly determined by joints 2 and 3, so the algorithm we provided has only these two inputs. Secondly, the height is determined only by joint 4, but this is not considered as the problem would be too complex, as we will see later on. Instead of considering values of joint 4 to 6 across the set of values they assume on the trajectory, we consider all possible values they can assume within their limits, even though this may result too conservative.

Off-line table computation

The off-line phase consists on building a table which contains all possible values of θ_2 and θ_3 as inputs and returns the rectangle which contains the robot once θ_1 is locked and according to all possible values of θ_4, θ_5 and θ_6. The on-line phase is then simplified to access the table and build a new rectangle which includes all the rectangles obtained with θ_2 and θ_3 across the set. Clearly the table is sampled to a subset of possible values of θ_2 and θ_3 and its size (and therefore its access time) is proportional to the sampling interval. This is also the reason why having more than two inputs has a lot higher computational cost. Hence we chose to sample the range by intervals of $5°$, which allows to contain the table size and to have enough resolution, without having to waste a large amount of volume.

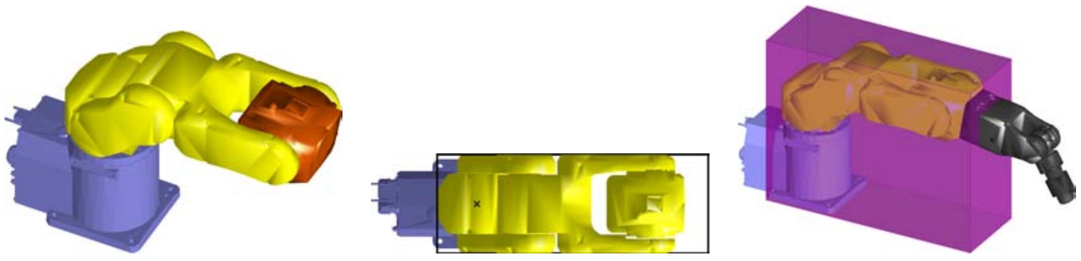
Since this stage is performed only once off-line and it is only related to the robot, we didn't care about finding an efficient solution but we applied a brute force implementation.

4.1.1.1 *Particularization to a subset of joints*

Despite the previous method can be efficient when the robot are far, it quickly fails when they are working a little bit closer. To still get the power of this solution we can use the same approach for a subset of the kinematic chain instead of the full robot, which at least lets us exclude part of the joints from collision detection in the next stage, preserving the speeding-up approach this technique gives (fig. 4.3c).

Reduced table

The idea is the same as the previous one, except that we do not have to care about the last joints when we build the table. In particular, we may be interested only in the sub-chain composed by the first 3 joints. In this occasion, the range of joint 3 can be neglected as it does not affect significantly the bounding box size. This is due to the



(a) Sub-chain made by first 3 joints. **(b)** Bounding Box for first 3 joints. **(c)** Bounding Box comparison with entire robot shape.

Figure 4.3: Bounding Box for sub-chain composed by first 3 joints.

shape of joint 3 (fig. 4.3a). In this way we can simplify the table letting it dependent only on θ_2, which reduces its access time. With this method we can achieve partial results which can be merged together, as it is explained in Section 4.3.

4.1.2 Swept Volume computation

Once we have retrieved the bounding box with joint 1 locked, we have to compute the sweep across the values assumed by joint 1. Since we treated the previous problem as planar, this situation will be handled as planar as well. The idea is to find a circular sector which contains all the rectangles obtained rotating them around the z axis of joint 1 by $\theta \in [\min(\theta_1), \max(\theta_1)]$. The center of this sector eventually will not be coincident with the rotational axis, since we must handle the rear and lateral obstruction of the robot. Hence, to completely describe the circular sector, we must calculate its center, its radius and the bounding angles (fig. 4.4).

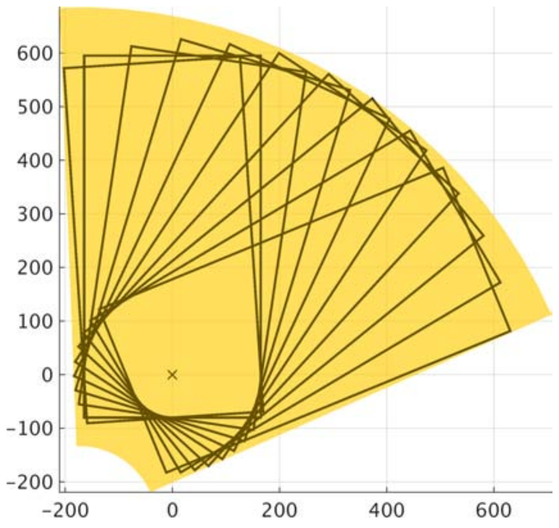


Figure 4.4: Robot bounding box sweep around joint 1 axis. This is the goal of this stage.

For the following procedure we will refer to the parameters shown in fig. 4.5, which can be straightforwardly obtained from the table lookup.

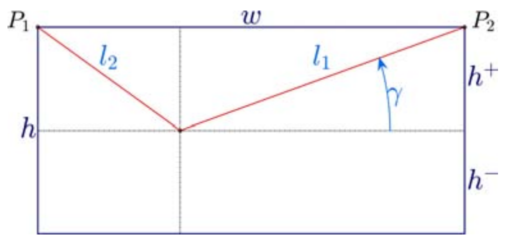
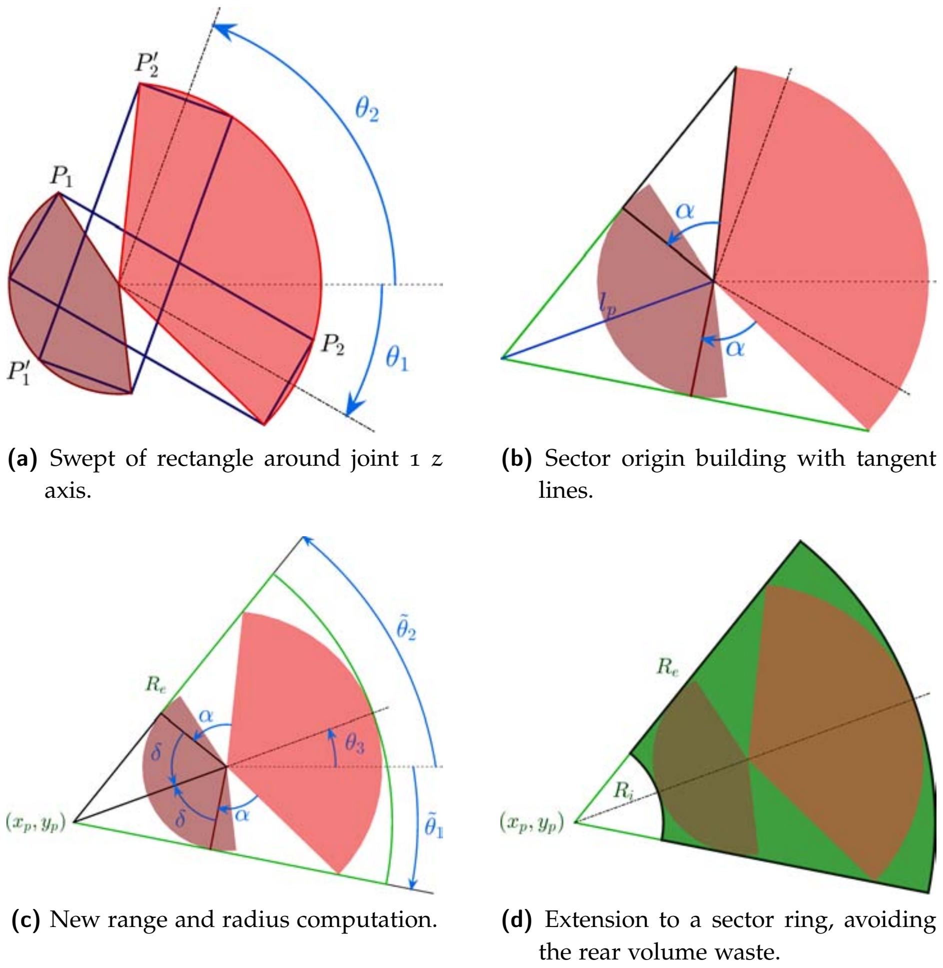


Figure 4.5: Parameters of rectangle given by table lookup.

For simplicity, during the table computation, we have imposed rectangle height symmetry, i. e., $h^+ = h^-$ (cfr. fig. 4.4), due to robot symmetry. However this can easily be removed and with only a few modifications to the next procedure. From now on, since we will be dealing with joint 1 only, we will denote with θ_1 and θ_2 the boundary angles of the range of the first joint.



(a) Swept of rectangle around joint 1 z axis.

(b) Sector origin building with tangent lines.

(c) New range and radius computation.

(d) Extension to a sector ring, avoiding the rear volume waste.

Figure 4.6: Robot bounding box building after swept plan view rectangle around joint 1 axis.

The first thing to notice is that P_1 and P_2 describe circle arcs as θ varies across its range (fig. 4.6a). The effective range we must take into account, due to the symmetry, is extended by 2γ. To build a circular sector which contains both sectors we must find the line which passes through P_2' and is tangent to the circle which describes the rotation of P_1 around $[0, 2\pi]$ (fig. 4.6b). We then do the same for the other boundary and then we find the intersection point between these lines, which will be the center of the sector. We must point out some facts:

Observations and method limits

- The previous construction is enough condition to contain the robot shape, since P_1 does not revolve around $[0, 2\pi]$, but only a sub-interval. However, since we can exploit the tangent, the procedure is simplified and it doesn't lead to a significant waste of volume on the final outcome (fig. 4.7);
- The tangent lines do not intersect when joint 1 does not move. In this case this stage can be skipped straightforwardly.
- The tangent lines could intersect on the front of the robot rather than behind it (where the front is on the right side of each figure). This happens when $l_2 > l_1$, which means the robot is performing a movement with major rear obstruction. In this case it is enough to swap l_2 with l_1;
- The tangent building is correct until the sweep is convex, i. e., until condition $\theta_2 - \theta_1 + 2\gamma < 180°$ is satisfied. This is not a problem as this technique is going to fail for large sweeps, so it is a good choice to split the trajectory into sub-paths as we did;
- Conversely, if the sweep is small the tangent lines are almost parallel and the radius of the sector is increasingly wide, wasting a lot of volume. For this reason we extended this technique to circular sector rings (fig. 4.6d).

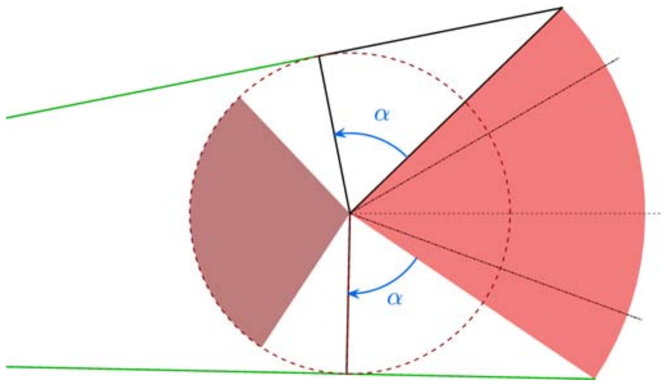


Figure 4.7: Sector boundary line building when the tangent point does not belong to the rear obstruction sector swept.

Swept volume computation procedure

With refer to fig. 4.6, we now summarize the main steps.

The black triangle highlighted in fig. 4.6b is rectangle and both the hypotenuse and the adjacent cathetus are known (they are l_1 and l_2 respectively). Therefore it is easy to find α:

$$\alpha = \arccos\left(\frac{l_2}{l_1}\right)$$

If we denote the extended range (highlighted in light red in fig. 4.6) as

$$\Delta\bar{\theta} = \theta_2 - \theta_1 + 2\gamma$$

we can define the mid-range angle θ_3 as

$$\theta_3 = \theta_1 - \gamma + \frac{\Delta\bar{\theta}}{2}$$

and, referring to fig. 4.6c, we obtain

$$\delta = \frac{360 - \Delta\bar{\theta} - 2\alpha}{2}$$

Once again, the black triangle in fig. 4.6c is rectangle so it is easy to find that

$$l_p = \frac{l_2}{|\cos\delta|} \qquad \text{and then} \qquad \begin{cases} x_p = -l_p \cos\theta_3 \\ y_p = -l_p \sin\theta_3 \end{cases}$$

Finally, according to fig. 4.6d,

$$R_e = l_p + l_1 \qquad \text{and} \qquad R_i = l_p - l_2$$

The new range is then

$$\Delta\tilde{\theta} = 180 - 2\delta \qquad \text{and so} \qquad \begin{cases} \tilde{\theta}_1 = \theta_3 - \frac{\Delta\tilde{\theta}}{2} \\ \tilde{\theta}_2 = \theta_3 + \frac{\Delta\tilde{\theta}}{2} \end{cases}$$

The extension to a circular sector ring will cost a few more steps during the collision detection phase which we will outline on the next section, so if the original range is quite wide we can neglect the last step and keep a simple circular sector.

The result of this modelling for the whole robot chain is shown in fig. 4.8. This is referred to a sample trajectory travelled by an Epson® C4A601S.

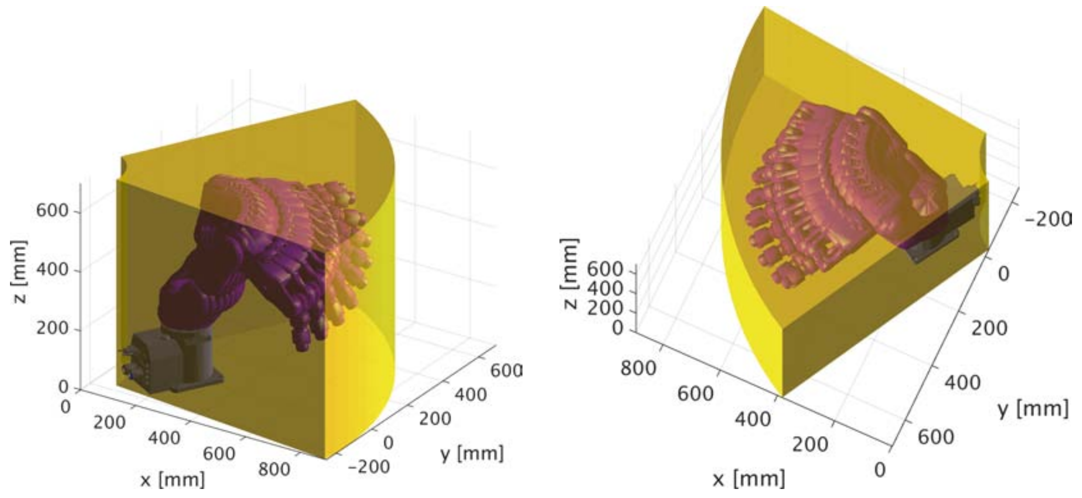


Figure 4.8: Robot swept for a sample trajectory and its bounding box sector built using the technique explained in this section (seen from two different views). Robot considered is a Epson® C4A601S.

4.1.3 Collision detection between robot swept volumes

Once we know the bounding volume of each robot swept, we must provide an algorithm to find whether two sweeps are intersecting. This means we must find a fast way to find the intersection between (in general) two circular sector rings. To do this, we adopted a straightforward method which considers the intersection between the boundaries of the sectors. This, of course, excludes the case when one is inside the other one, but this is not meaningful as it cannot happen in practice. As long as there are two curved lines and two straight lines for each sector boundary, we must provide three stages which test the intersection between them.

Since we are dealing with two different reference systems, we are using two polar representations of a circle, assuming we know the relationship between them (which we do, as we know the workcell configuration).

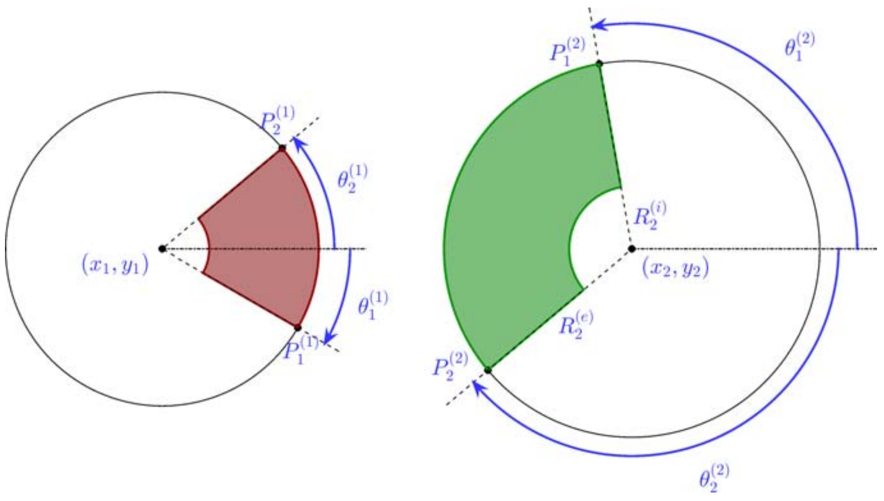


Figure 4.9: Parameters of two circular sector rings with their reference systems. Superscript is for reference system, while subscript refers to each boundary of the sector.

Intersection-free enough condition

Before testing intersection, the first thing to do is to test whether the two circles of radii $R_e^{(1)}$ and $R_e^{(2)}$ are intersecting, which means that if

$$d \triangleq \mathrm{dist}\big((x_1, y_1), (x_2, y_2)\big) > R_e^{(1)} + R_e^{(2)} \tag{4.1}$$

there is no intersection and we can skip all the remaining steps (fig. 4.9).

4.1.3.1 *Intersection between segments*

We start by defining the boundary points on the external circles in polar coordinates:

$$P_i^{(1)} : \begin{cases} x_{p_i}^{(1)} = x_1 + R_e^{(1)} \cos\theta_i^{(1)} \\ y_{p_i}^{(1)} = y_1 + R_e^{(1)} \sin\theta_i^{(1)} \end{cases} \qquad i = 1, 2 \tag{4.2}$$

$$P_j^{(2)} : \begin{cases} x_{p_j}^{(2)} = x_2 + R_e^{(2)} \cos\theta_j^{(2)} \\ y_{p_j}^{(2)} = y_2 + R_e^{(2)} \sin\theta_j^{(2)} \end{cases} \qquad j = 1, 2 \tag{4.3}$$

The boundary segment for each circular sector ring is then defined as

$$\begin{cases} x^{(i)}(s) = x_1 + s\left(x_{p_i}^{(1)} - x_1\right) \\ y^{(i)}(s) = y_1 + s\left(y_{p_i}^{(1)} - y_1\right) \end{cases} \qquad i = 1,2,\ s \in \left[\frac{R_i^{(1)}}{R_e^{(1)}}, 1\right] \tag{4.4}$$

$$\begin{cases} x^{(j)}(t) = x_2 + t\left(x_{p_j}^{(2)} - x_2\right) \\ y^{(j)}(t) = y_2 + t\left(y_{p_j}^{(2)} - y_2\right) \end{cases} \qquad i = 1,2,\ t \in \left[\frac{R_i^{(2)}}{R_e^{(2)}}, 1\right] \tag{4.5}$$

A segment of the first sector (4.4) will intersect a segment of the second one (4.5) only if the systems (4.4) and (4.5) are simultaneously satisfied within their domains.

Let's suppose no segment pair is parallel. Then, after some simple steps, we get

$$\begin{cases} \bar{t} = \dfrac{\left(y_1 - y_2\right)\left(x_{p_i}^{(1)} - x_1\right) + \left(x_2 - x_1\right)\left(y_{p_i}^{(1)} - y_1\right)}{\left(y_{p_j}^{(2)} - y_2\right)\left(x_{p_i}^{(1)} - x_1\right) - \left(x_{p_j}^{(2)} - x_2\right)\left(y_{p_i}^{(1)} - y_1\right)} \\ \bar{s} = \dfrac{\left(x_2 - x_1\right) + \bar{t}\left(x_{p_j}^{(2)} - x_2\right)}{x_{p_i}^{(1)} - x_1} \end{cases} \qquad i,j = 1,2 \tag{4.6}$$

If two segments are parallel, then the denominator on the first of (4.6) is zero and clearly that solution must be dropped. The previous system has at most 4 solutions, one for each pair of segments. The condition for intersection is then

$$(\bar{s}, \bar{t}) \in \left[\frac{R_i^{(1)}}{R_e^{(1)}}, 1\right] \times \left[\frac{R_i^{(2)}}{R_e^{(2)}}, 1\right] \tag{4.7}$$

4.1.3.2 *Intersection between circles*

If (4.7) is not satisfied, we proceed finding the intersection between each pair of circles (both the internal and the external ones which define each ring, if present). First of all, we define the mid-range angle between the 2 sectors (fig. 4.10):

$$\theta_0 = \arctan2(y_2 - y_1, x_2 - x_1)$$

As can be seen in fig. 4.10, this allows to find only 2 angles instead of 4, due to symmetry:

$$\delta_1 = \arccos\left(-\frac{R_2^2 - R_1^2 - d^2}{2R_1 d}\right) \tag{4.8}$$

$$\delta_2 = \arccos\left(\frac{R_1^2 - R_2^2 - d^2}{2R_2 d}\right) \tag{4.9}$$

Referring to the original systems (fig. 4.11), these lead to

$$\begin{cases} \alpha_1^{(1)} = \theta_0 + \delta_1 \\ \alpha_2^{(1)} = \theta_0 - \delta_1 \\ \alpha_1^{(2)} = \theta_0 + \delta_2 \\ \alpha_2^{(2)} = \theta_0 - \delta_2 \end{cases} \tag{4.10}$$

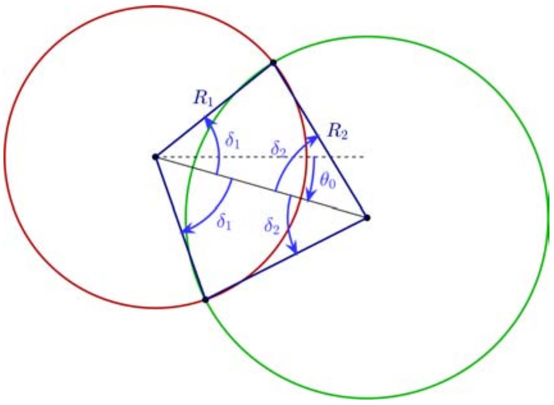


Figure 4.10: Intersection parameters between circles.

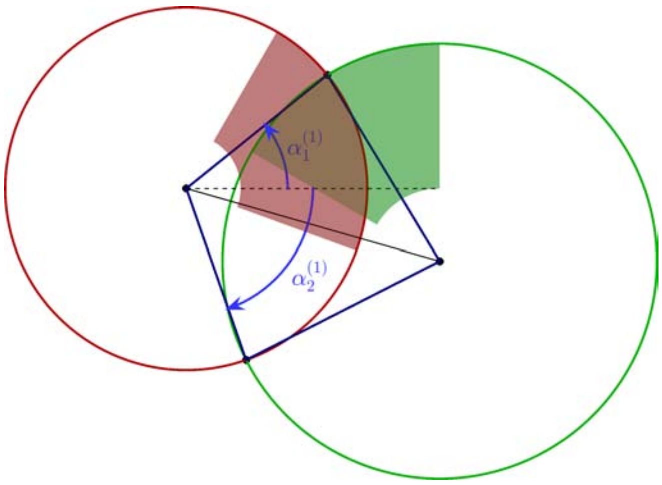


Figure 4.11: Intersection between circles and sector range check.

As highlighted on fig. 4.11, the intersection points may not lie within the range of each sector. Since angles evaluated in (4.10) are referred to the original systems, this check is straightforward[2], so there is intersection only if one of the following is satisfied:

$$\alpha_1^{(1)} \in \left[\theta_1^{(1)}, \theta_2^{(1)}\right]$$
$$\alpha_2^{(1)} \in \left[\theta_1^{(1)}, \theta_2^{(1)}\right]$$
$$\alpha_1^{(2)} \in \left[\theta_1^{(2)}, \theta_2^{(2)}\right]$$
$$\alpha_2^{(2)} \in \left[\theta_1^{(2)}, \theta_2^{(2)}\right]$$

4.1.3.3 *Intersection between circle and segment*

To avoid any coordinate system conversion, we keep each segment as described in polar coordinates. In the following procedure, we use the subscript 2 for the circle

2 every time there is an in-range check it is necessary to pay attention to modulo problems.

and 1 for the segment, which refers to θ_1 (fig. 4.12). This procedure must then be

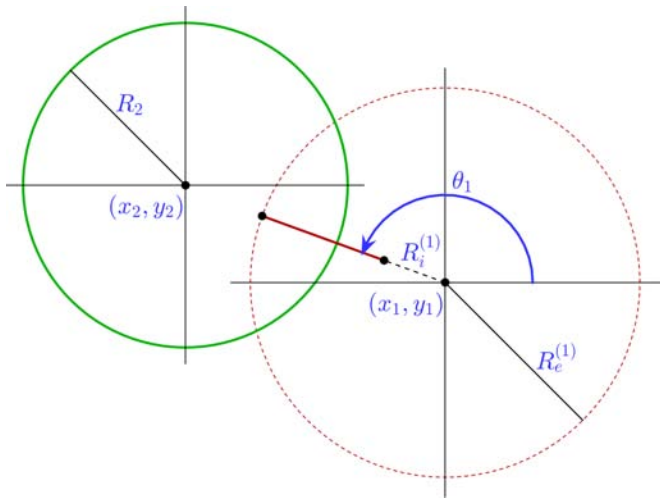


Figure 4.12: Intersection parameters between circle and segment.

performed for each boundary segment of each sector matching the other circles, i. e., it has to be done at most 8 times, swapping the rule of circle and segment. The circle and the segment are parametrized as follows:

$$C_2 : \begin{cases} x(\theta) = x_2 + R_2 \cos\theta \\ y(\theta) = y_2 + R_2 \sin\theta \end{cases} \qquad \theta \in \left[\theta_1^{(2)}, \theta_2^{(2)}\right] \tag{4.11}$$

$$S_1 : \begin{cases} x(\rho) = x_1 + \rho \cos\theta_1 \\ y(\rho) = y_1 + \rho \sin\theta_1 \end{cases} \qquad \rho \in \left[R_i^{(1)}, R_e^{(1)}\right] \tag{4.12}$$

Again, the intersection is given by any common solution between (4.11) and (4.12) within the specified ranges. Defining the transformation[3]

$$u \triangleq \tan\left(\frac{\theta}{2}\right) \qquad \text{such that} \qquad \begin{cases} \cos\theta = \frac{1-u^2}{1+u^2} \\ \sin\theta = \frac{2u}{1+u^2} \end{cases} \tag{4.13}$$

we obtain, after few steps

$$\rho\cos\theta_1 = (x_2 - x_1) + R_2 \frac{1-u^2}{1+u^2} \tag{4.14}$$

$$\rho\sin\theta_1 = (y_2 - y_1) + R_2 \frac{2u}{1+u^2} \tag{4.15}$$

If $\theta_1 = \pm\frac{\pi}{2}$, we get u from (4.14) and we solve directly (4.15) in ρ. If $\theta_1 = k\pi \; (k = 0, 1)$ we do the opposite, skipping the next stage which involves tangent and cotangent computation. Let's now consider the condition

$$\theta_1 \in \left[\frac{\pi}{4}, \frac{3\pi}{4}\right] \cup \left[-\frac{3\pi}{4}, -\frac{\pi}{4}\right] \setminus \left\{\pm k\frac{\pi}{2},\ k \in \mathbb{Z}\right\} \tag{4.16}$$

3 the inverse transform may have some singularity problems, but these do not warn us as we will see later on

If (4.16) is met, we can get ρ from (4.14) and solve (4.15), which, after simple steps, leads to

$$\begin{gathered}\Big[(x_2-x_1)\tan\theta_1 - R_2\tan\theta_1-(y_2-y_1)\Big]u^2-2R_2u+\\+\Big[(x_2-x_1)\tan\theta_1+R_2\tan\theta_1-(y_2-y_1)\Big]=0\end{gathered} \tag{4.17}$$

avoiding any numerical problem due to the discontinuity of the tangent. If (4.17) does not have any solution, there is no intersection between this particular circle/segment pair. If there exist any solution $\bar{u}$, then we have

$$\bar{\theta}=2\arctan\left(\frac{1}{2}\frac{\bar{u}}{(x_2-x_1-R_2)\tan\theta_1-(y_2-y_1)}\right) \tag{4.18}$$

which gives

$$\bar{\rho}=\frac{y_2-y_1+R_2\frac{2\bar{u}}{1+\bar{u}^2}}{\sin\theta_1}$$

If the denominator in (4.18) is 0, we simply skip the arctangent computation since $\bar{\theta}=\pi$ and we are not interested in its sign.

Conversely, if (4.16) does not hold, we can still get ρ from (4.15) and solve (4.14); after an analogous procedure, we get

$$\bar{\rho}=\frac{x_2-x_1+R_2\frac{1-\bar{u}^2}{1+\bar{u}^2}}{\cos\theta_1}$$

Regardless of the method used, the condition for intersection is

$$(\bar{\rho},\bar{\theta})\in\left[R_i^{(1)},R_e^{(1)}\right]\times\left[\theta_1^{(2)},\theta_2^{(2)}\right] \tag{4.19}$$

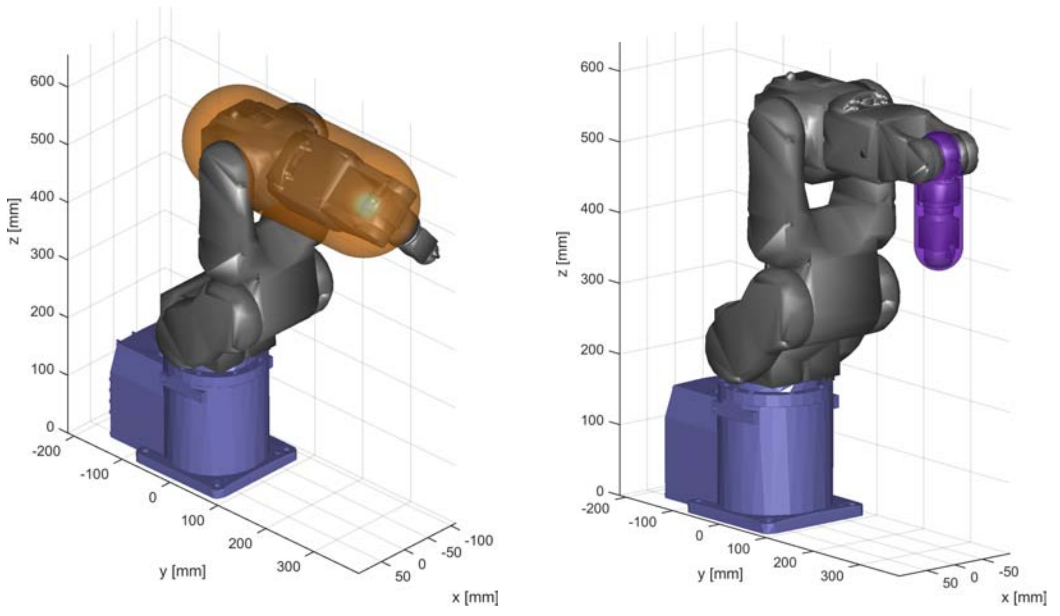
Since this is the last step, if no intersection is found after all circle/line combinations, then there is no intersection between the paths considered.

4.2 Group of Joints approach

The aim of the previous technique is to avoid collision detection algorithm presented on Chapter 3 whenever possible, since some checks across time and across space (i. e., joint modelling) might be unnecessary. When this step fails, however, the computational load remains a problem. For example, if we are dealing with two anthropomorphous manipulators with 6 axes each and we analyse only the load given by SSV algorithm, we would still have to perform *LL algorithm* at least $36 \times N_1 \times N_2$ times, whereas if it succeeds none of them is needed. This led us thinking about an intermediate stage which aimed to reduce the number of calls to the lower-level hybrid algorithm developed in Chapter 3.

At this stage we cannot do any more about the robot sweep across time, so we developed this level working only on a minor subdivision of the robot global shape. The reason of this choice is motivated by the fact that if we can reduce the number of entries to feed to hybrid algorithm, the number of iterations is reduced exponentially.

For instance, a structure with only 4 elements leads to 16 iterations for each time pair, which is less than half compared to the original check. The drawback is that this is an additional test which does not substitute the one developed in Chapter 3 in case of failure. Once again, this improvement has been tested only on an Epson® anthropomorphous robot, even though this can be extended to most manipulators of the same class as the robot shape is usually similar.



(a) Links 3 and 4 **(b)** Links 5 and 6

Figure 4.13: Joint SSV groups formed by contiguous links of an Epson® C4A601S.

Robot modelling with fewer structures

The idea is that some contiguous joints share the same rotational axis and their radial size is almost the same. This is common on joint pairs $3-4$ and $5-6$ of the Epson® C4A601S (fig. 4.13). Moreover, we may have modelled some joints with more SSVs or with a polygonal SSV, so it is worth to anticipate the final test with a rawer model, which is faster. As a result, we modelled the robot with fewer SSVs than already done, grouping consecutive joints which share the z axis and anyway using no more than one linear SSV for the other joints (fig. 4.14).

In this occasion we did not provide the user with a suitable interface, but we only gave him the possibility to specify the tolerance of the radii of the joint groups, while the single joints (like links 1 and 2 on fig. 4.14) inherit the parameters from a basic configuration file, which can be processed with the GUI provided to build usual SSVs. The way to get new SSV parameters is the same used for single joints and it is explained in Section 3.3.

Collision detection between simplified robot structures

Once the modelling phase has been completed, the on-line step is identical to a normal collision detection phase between two linear SSVs, which employs *LL-algorithm*. If the outcome of this check is collision presence, we must perform collision detection hybrid algorithm as usual (this time between only the joints interested in the block), otherwise we can skip any further test for the joints interested in the block for the current time.

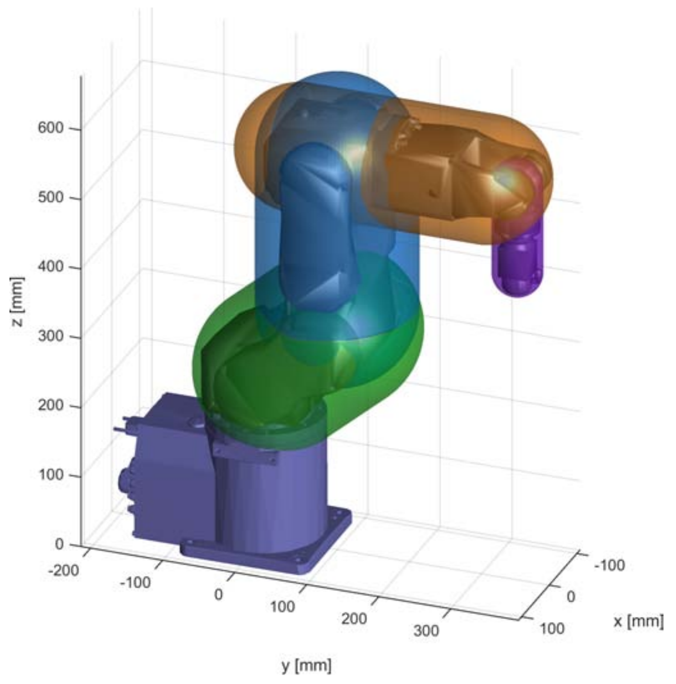


Figure 4.14: Joints SSV grouping for an Epson® C4A601S robot.

4.3 Algorithm outline

4.3.1 Trajectory subdivision

The previous algorithm can be applied to the whole trajectory, but in case of failure we will lose all its advantages. Moreover, there are some areas where one or both the robots can work safely, which more likely could be the workspace for only part of the trajectory. For this reason, a smart choice is to split the trajectory into disjointed sub-paths and then iterate this algorithm for all of these.

This split must be performed on-line, so it must be very fast. As said at the beginning of this chapter, the easiest thing to do is to split the trajectory into sub-intervals according to the first joint position. Even though fast speed-sign variation are not very common during a PTP motion (cfr. Chapter 2) for joint 1, we decided to apply a hysteresis thresholding to the position trend (fig. 4.15), in order to avoid any undesirable dense subdivision.

Thresholding pros and cons

The threshold interval remains set by the user and its value can affect significantly the fastness of the algorithm:

- ◇ A small threshold guarantees maximum efficiency of the optimized algorithm, as parts of the trajectory where there is no collision are usually solved prior to the lower-level hybrid algorithm. However, the cost to compute the robot sweep and the intersection between sweeps can raise significantly.
- ◇ A high threshold lowers the time required to the overhead (robot sweep computation and intersection test) and it reduces the time required in case of collision-free trajectories, but it loses its advantages as soon as there is intersection be-

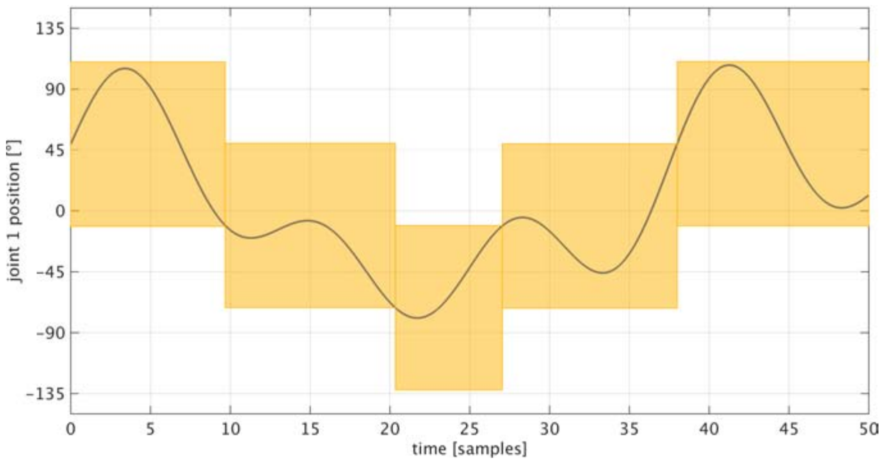


Figure 4.15: Hysteresis threholding for a sample signal.

tween the sweeps. Moreover, the thresholding must not exceed a certain value as the sweep might become concave.

To avoid volume wasting, the robot sweep is performed according to the maximum and the minimum value of θ_1 within the specific time window and not to the actual threshold values.

4.3.2 Combination between different abstraction levels

Sub-chains used in the final algorithm

As we said, we can model the robot sweep using sub-chains as well. Hence, we can perform multiple stages during sweep intersection check, in order to exclude at least part of the robot from the next low-level collision detection. This could be applied to all possible sub-chains that start from joint 1 obtainable from the robot. However this could be uselessly expensive as the information carried by the new sub-chain might hardly exclude sub-chains from the collision detection phase. For example, the bounding box which considers the sub-chain composed by joints 1-2-3 of an Epson® C4A601S is almost the same size as the sub-chain 1-2 (fig. 4.3c on page 68), so it is not worth to perform any check with the last one, as it likely produces the same results as the first one, which, in case of success, can exclude more joints form the next step. The same result is visible for sub-chain which excludes only last joint and the gripper.

As a result, we will only use models which consider sub-chains:

- from joint 1 to 3, depending only on joint 2;
- from joint 1 to 4, depending on joint 2 and 3;
- from joint 1 to 6, depending on joint 2 and 3.

The previous sub-chains will be built as described in Section 4.1.1. These models will be integrated in the algorithm explained in next section.

4.3.3 High-level collision-free algorithm

At this point we developed an algorithm, whose steps are better shown as a state diagram, like it has been done for the collision detection algorithm (fig. 4.16). This stage is performed for each couple of time windows of each robot, as determined by trajectory split discussed before.

The idea behind it is to avoid calls to the lower-level hybrid algorithm as many times as possible. We start testing the intersection between the entire robot sweeps (indicated as $6, 6$ with orange background on fig. 4.16). If there is no collision, we can skip to the next pair of sub-paths, otherwise we have to perform collision detection between the block containing the last 2 joints of each robot, as there is no other way to test their collision at high level. In fact, eventually we must be allowed to perform collision detection tests between every possible pair if needed, as this must produce a sufficient condition for collision-free paths.

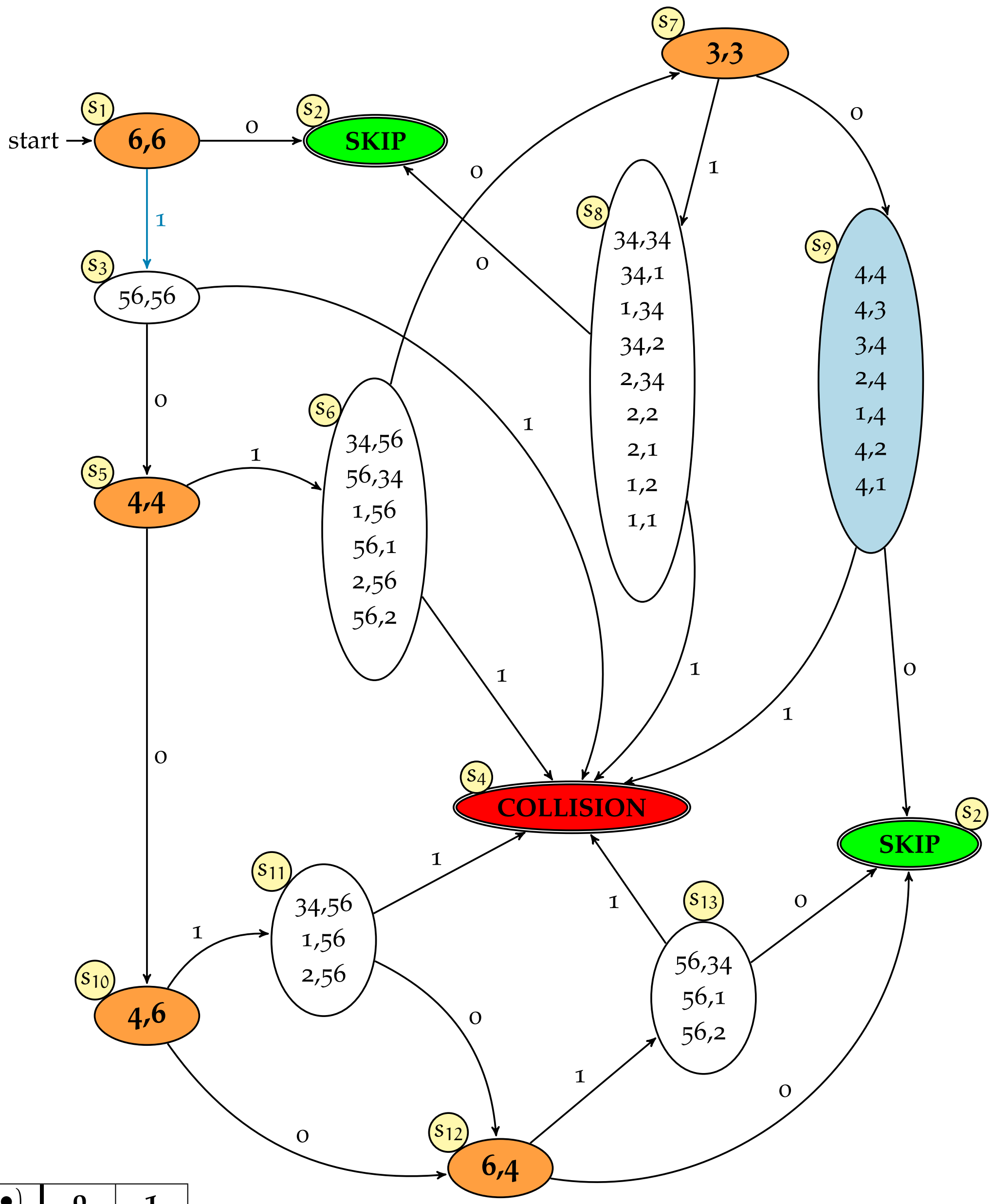
We must point out that the collision state in fig. 4.16 indicates a real collision, so any state with no background means at that stage we must first of all check collision between the macro-groups (as explained in the previous section) and, if they intersect, the hybrid algorithm must be invoked, inheriting the final outcome from there.

Kinematic model storage

The highlighted transition exiting from state 1 means that if the algorithm follows that route we must update and store the kinematic model of each robot for all the time pairs belonging to the respective time windows. This is because certainly we will have to perform either SSV or Box algorithm for each time pair within the time windows. Therefore it is advisable to store the transformation matrices $\mathbf{T}_{ia}$ as they are likely to be needed more than once.

At this point, if no collision has been detected, we try to exclude at least the first 4 joints from the low-level collision detection (4,4). In case of intersection, we try to do the same with sub-chain 1-2-3 (state 7), after having checked collision between the joints which remains outside of this sub-chain. If, again, we find collision, we give up with the robot sweep intersection test and we end up with the last macro-group of joints remaining. Conversely, if sub-chains 1-2-3 do not collide but 1-2-3-4 do, we skip the macro-group test and we go straightforwardly to the low level algorithm (the blue state in fig. 4.16). If the first 4 joints do not collide with each other, instead, we try to compare different-length chains (4,6 and then 6,4) and then we follow the same idea as before in case one of them fails.

On fig. 4.16 more details are explained.



$\delta(s,\bullet)$	0	1
s_1	s_2	s_3
s_2	s_2	s_2
s_3	s_5	s_4
s_4	s_4	s_4
s_5	s_{10}	s_6
s_6	s_7	s_4
s_7	s_9	s_8
s_8	s_2	s_4
s_9	s_2	s_4
s_{10}	s_{12}	s_{11}
s_{11}	s_{12}	s_4
s_{12}	s_2	s_{13}
s_{13}	s_2	s_4

Figure 4.16: Automaton which represents all the states of the high-level algorithm and its state transition function. Transitions labelled with "1" mean that at the previous stage a collision was detected, while a "0" means that there is no collision between the joints interested or the sub-chain tested. State 2 is replied twice for sake of clarity. Terminal states transition are not reported as the algorithm stops whenever it reaches one of these states. For each state, the comma separates the joints of the first robot (on the left) with the ones of the second robot. White states represent collision detection tests between macro-groups of joints, for example 34 indicates the block composed by joints 3 and 4. The blue labelled state means that the joints considered are single joints and not macro-groups. Orange-labelled states stand for robot sweep intersection stage between the sub-chains which terminate with the joint indicated.

5

Experimental results

Here we show the main results obtained with collision detection algorithm, trying to outline the practical meaning of parameters involved and to produce a suitable set of options that allows to reach the best results.

5.1 Reliability and precision of the algorithm

Tolerance of link representation

First of all, we must point out that the main requirement for this project was the fastness of the final algorithm. As we showed, the final criteria which defines collision presence is made by the representation of each robot link by an 18-DOP or a 30-DOP. The reliability and the degree of precision of our method are then inherited from the precision of this modelling phase, which was previously done and is not part of this work. Clearly, the modelling phase itself has some parameters which can be tuned according to the user requirements, of which the main one probably is the tolerance. As said in Section 3.3, SSV models can be made from points given by Box modelling, so if we want to increase the tolerance we only have to update the 18-DOP model.

This is an important point and it links the work done for the trajectory identification with the second part. In fact the trajectory which we are feeding into collision detection algorithm is affected by some error, which we do not directly take into account on the algorithm. The only way we can consider the estimation error is by increasing the model tolerance by a value which ensures to take into account of maximum error allowable on the estimation phase. By the way, the tolerance set up does not affect directly the collision detection algorithm performance, instead it regulates the false alarm rate. In fact, the higher is the tolerance imposed, the more volume is wasted for a link model and then the more probable is the event that the robot aren't colliding but their models do. Therefore it is important to keep the maximum allowable error of the estimated path as low as possible.

Point Cloud modelling

There can be another abstraction level which can be used, which is even potentially more accurate than 18 or 30-DOP modelling, which uses point cloud robot modelling, as explained at the beginning of Chapter 3. We did not exploit this method, as it is computationally expensive, therefore it shall be used when execution time is not critical, for example when two robots are working in strict cooperation. Moreover, since the tolerance to be imposed is not negligible, this modelling would be pointless as its degree of precision lies on the knowledge of the exact robot position, i. e., at this point it would not give more reliable information compared to a 30-DOP model.

5.2 Implementation details

Performance with unoptimized algorithm

During the developing phase we implemented all the code with MATLAB® . After the first part of the algorithm (the one explained in Chapter 3) was completed, we tested it and came up with some performance results. As repeatedly said, at this level the worst case is when two robots are very far from each other, especially if joints are modelled with more SSVs or with polygonal SSVs. First results showed that the execution time for a simple movement made by 50 knots for each robot lasted for about 74 seconds. There are however even worse situations when there is no collision but the robots are close enough to step down to level 2 or 3 (as indicated in fig. 3.19[1]) because SSV modelling is not precise enough. This step is only rising the execution time, so that for another sample movement in which this condition is met collision absence is found after 82 seconds, with a total of 131 steps through level 2 or 3. However, SSV modelling allowed to gain about a 50× speed-up factor compared to Box modelling only (i. e. using automaton 2 in fig. 3.18).

Final implementation details: Matlab vs C

These results are far from the target, which is set to be within 0.1 s. This is the reason why we developed all the following part. After the completion of the entire algorithm, we decreased to about 5 seconds in the worst case, which was way faster than the previous approach. After some investigations, we realised that most of the time was spent in overhead by MATLAB® , so we decided to gradually switch the bottleneck into a C implementation performed by some MEX (*Matlab EXecutable*) code. The final implementation regulates all the off-line setup and the outer cycle in MATLAB® , including the trajectory split and the intersection between circle sectors, whereas the inner cycles that perform SSV collision detection using *LL-algorithm* for joint blocks and hybrid algorithm are written in C. This solution allowed us to reach the final target, as we will discuss from the next section.

A final note to consider is that all the performance are given using MATLAB® timer and iterating the entire algorithm for 50 times with the same inputs, which ensures to not consider any untrusted value. All the processing is made with an Intel® Core™ i5 3317U processor with 4 1.70 GHz CPUs and 6 GB of RAM.

5.3 Modelling Set-up

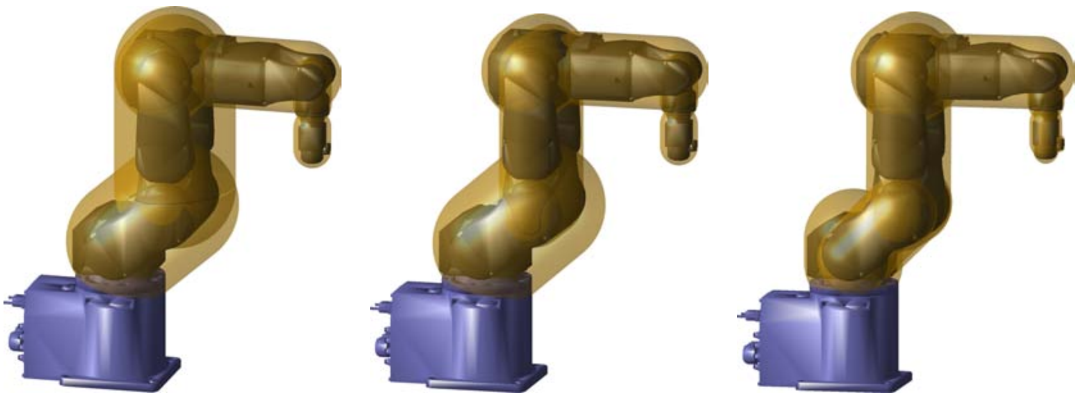
As there are three different types of robot modelling involved, we show some results obtained varying the available parameters.

5.3.1 SSV modelling

The user is provided with a GUI which allows to tune a lot of different settings. All these details are reported in Section 3.3. Eventually, there can be different situations.

The simplest one is when the robot is represented with all linear SSVs (fig. 5.1a). As previously said, this ensures the fastest precessing, but limited to this level. The

1 if we suppose to use that precise automaton

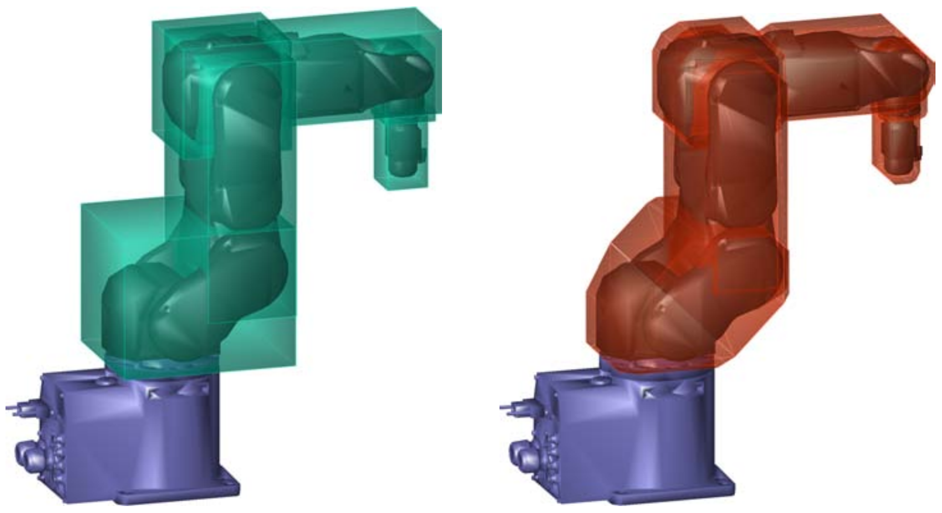


(a) SSV modelling with one linear SSV per joint. **(b)** SSV modelling with joint 2 represented with a rectangular SSV. **(c)** SSV modelling with more linear SSVs per joint.

Figure 5.1: Robot SSV modelling with different precision levels. Robot in question is an Epson® C4A601S.

problem is that, in case of collision, the next step might not reveal any true collision. This might be avoided a more precise model, like the ones in fig. 5.1b and 5.1c, which however may represent a waste of time when there is no collision.

5.3.2 Box modelling



(a) Box modelling with OBB. **(b)** Box modelling with 18 - DOP.

Figure 5.2: Robot modelling with bounding volumes. Robot in question is an Epson® C4A601S.

As our work didn't include this modelling, which however has been done in [10], we report only the models we are referring to. For the simple box modelling made by parallelepipeds we referred to an *Oriented Bounding Box* (OBB), which is the primitive for the detailed box. This is shown in fig. 5.2a.

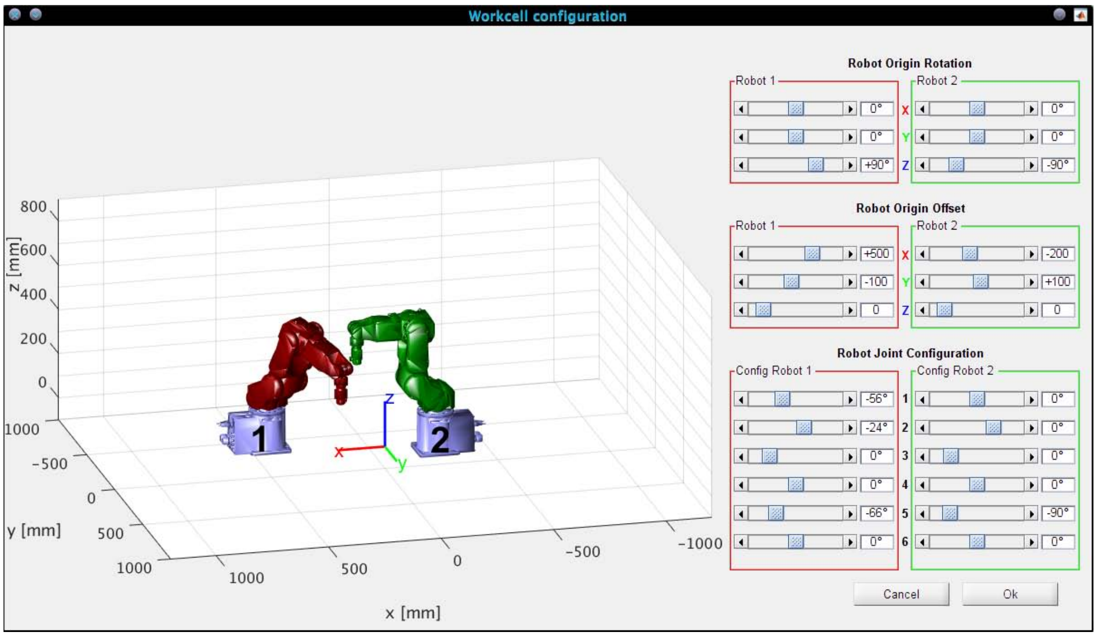
On the other hand, the detailed box uses an 18-DOP obtained from the OBB which contains each joint (fig. 5.2b). All the procedure is valid for a 30-DOP as well, but since we don't have a precise trajectory estimation, the level of precision given by a 18-DOP is enough, whereas a more complex model would increase processing time.

5.4 Critical situation test

To test the performance of our algorithm, we had to chose among many possible situations. Since we want to be as conservative as possible, we fed the algorithm with the most critical trajectories possible.

First thing to say is that all data we will be using are not real trajectory knots but simulated ones. Moreover they do not come from the previous estimation phase but from a simple uniform quantization of movements between effective points, i. e., there is no CP or Arch option involved. This is because we did not complete the whole estimation phase (as that is part of another thesis work). Anyway, the results interpretation is the same if the trajectory is the real one.

To produce every critical situation we used a simple GUI which allowed to move and rotate each robot base within the workcell, with the possibility to show the robot displacement given a certain configuration (fig. 5.3).



Robot Origin Rotation

Robot 1		Robot 2
0°	X	0°
0°	Y	0°
+90°	Z	-90°

Robot Origin Offset

Robot 1		Robot 2
+500	X	-200
-100	Y	+100
0	Z	0

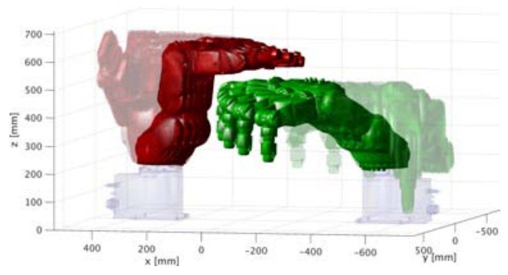
Robot Joint Configuration

Config Robot 1		Config Robot 2
-56°	1	0°
-24°	2	0°
0°	3	0°
0°	4	0°
-66°	5	-90°
0°	6	0°

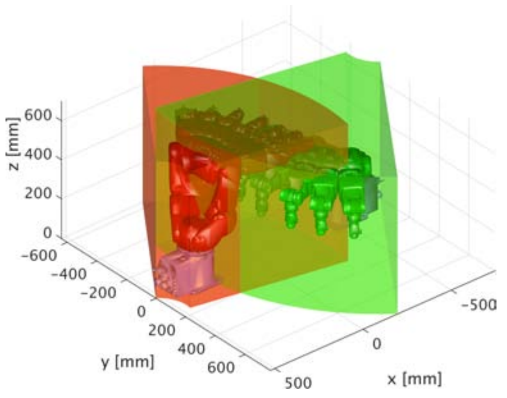
Figure 5.3: Simple GUI to create different workcell configurations with two robots.

A critical situation example

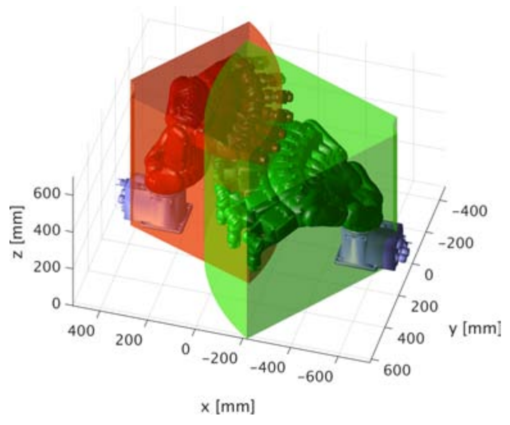
One of the most critical situations is depicted in fig. 5.4a. We can see the robot sweeps are very close to each other. To quantify the complexity of the situation we can refer to the automaton in fig. 4.16. In general, the more steps are required to get to a final state, the more time is required for the whole computation. The most critical part of the trajectories are the middle ones. In this occasion, test between entire robot



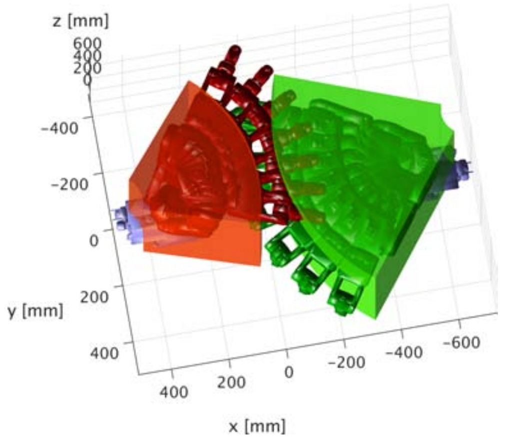
(a) Critical robot sweeps which are very close to each other.



(b) Sweeps modelled from all robot kinematic chains.



(c) Sweeps modelled from only first 4 joints.



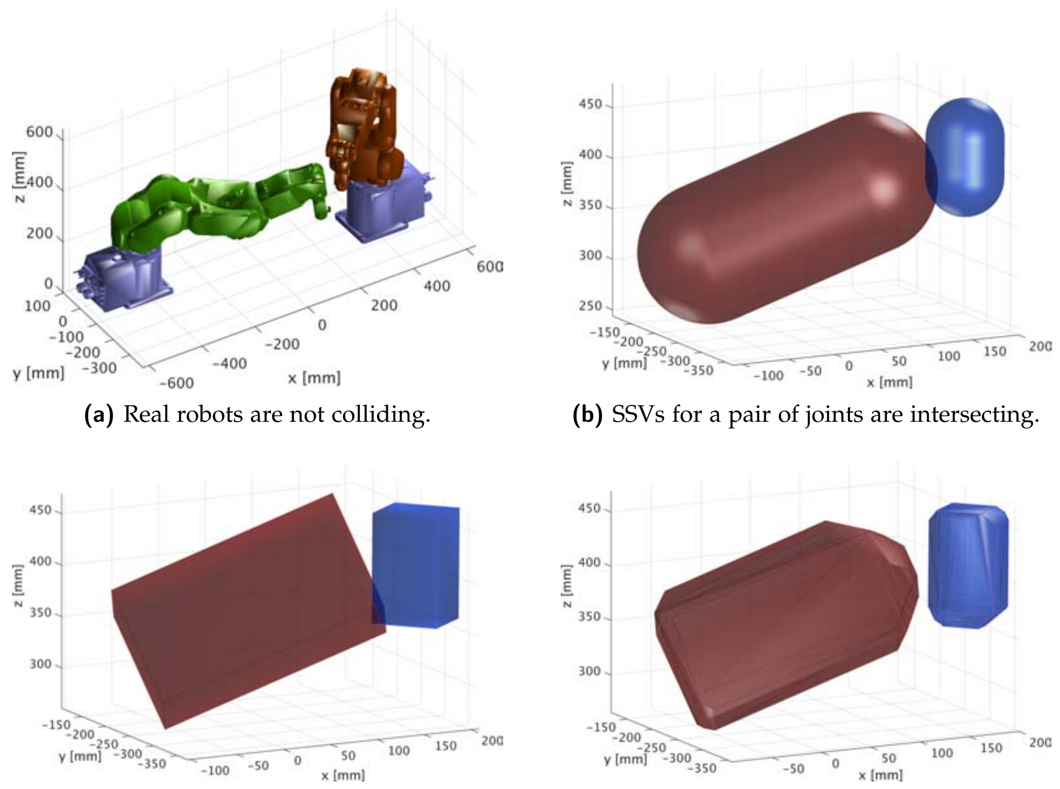
(d) Sweeps modelled from only first 3 joints. Intersection is not detected at this level.

Figure 5.4: Critical situation handling with sector-modelled sweeps.

sweeps (for the middle part of the trajectories) is going to fail[2] (fig. 5.4b), as well as the test between first 4 joints sweep (fig. 5.4c). After having tested joint blocks present in states s_3 and s_6 (cfr. fig. 4.16 on page 81) without any success, finally no intersection is found between robot sweeps made by first 3 joints (fig. 5.4d). This represents the power of this strategy, as it can avoid to check collision between any combination of first 3 joints for all the trajectory slice we are considering, which is a 25% load reduction, even in this critical situation. The last step is then performed by checking collision presence between joints considered by state s_9 with hybrid algorithm, as described in Chapter 3.

In this particular occasion, precision of SSV modelling may not be enough accurate and Box algorithm may be repeatedly invoked to find whether there is collision between certain pair of joints (fig. 5.5). This can lead to long processing time due to the complexity of interpenetration between boxes algorithm (cfr. Section 3.2.1).

2 i. e., it triggers intersection between sectors, which means it needs a further investigation.



(a) Real robots are not colliding.

(b) SSVs for a pair of joints are intersecting.

(c) OBBs for the same pair of joints are intersecting.

(d) 18-DOP are not intersecting instead, which means no collision is found.

Figure 5.5: Collision detection with SSV and Box modelling in a critical situation, handled only at level 3.

In situations like this one, this represents the main load, which is why we tested our algorithm with these trajectories. However, if this is limited to only few couple of joints, execution time can remain within the limit, as it is on this case.

5.5 Simulation results

We tested the final algorithm with many situations, including other critical ones. Since there are many parameters to be tuned, we summarized some results with different set-ups, in order to choose the best ones. All results are shown in Table 2, where main fields are:

- config: it represents a certain workcell configuration and a specific trajectory. This defines some groups of simulations where we only changed collision detection parameters;
- collision: *NC* means that trajectories did not collide, *C* means they did;
- N_1, N_2: they are the number of knots for each trajectory;
- SSV R_1 and R_2 models: they indicate a specific SSV model for each robot:
  - B means a *basic* model with one linear SSV for each joint (fig. 5.1a);

ID	config	collision	N_1 [knots]	N_2 [knots]	SSV R_1 model	SSV R_2 model	aut	thr [°]	time [ms]
1	A	NC	50	50	B	B	3	140	**17.0**
2	A	NC	50	50	B	B	3	90	**29.7**
3	B	NC	51	54	B	B	3	140	**69.6**
4	B	NC	97	106	B	B	3	140	**141.6**
5	B	NC	51	54	B	B	3	90	**102.6**
6	B	NC	51	54	1P	1P	3	140	**70.0**
7	B	NC	51	54	1P	1P	1	140	**70.0**
8	B	NC	51	54	1P	1P	2	140	**75.3**
9	B	NC	51	54	PM	PM	3	140	**69.6**
10	C	NC	53	44	B	B	3	140	**41.6**
11	C	NC	113	94	B	B	3	140	**82.9**
12	C	NC	53	44	1P	1P	3	140	**41.4**
13	C	NC	53	44	1P	1P	1	140	**41.6**
14	D	C	50	50	B	B	3	140	**9.8**
15	D	C	50	50	B	B	3	90	**15.2**
16	D	C	50	50	1P	1P	3	140	**9.8**
17	D	C	50	50	1P	1P	1	140	**10.3**
18	D	C	50	50	B	B	4	140	**9.7**
19	E	NC	50	50	B	B	3	140	**37.9**
20	E	NC	50	50	1P	1P	3	140	**37.9**
21	F	C	51	54	B	B	3	140	**37.2**
22	F	C	51	54	1P	1P	3	140	**37.8**
23	G	NC	52	52	B	B	3	140	**58.8**
24	G	NC	52	52	1P	1P	3	140	**57.3**

Table 2: Resuming table with main results obtained during collision detection phase. Red-highlighted cells mean that time requirement is exceeded. Yellow and blue rows represent two groups of simulations with the same parameters but different workcell configuration and trajectories.

  - 1P means there is only one polygonal joint (for joint 2) and the other ones are linear (fig. 5.1b);
  - PM means there is one polygonal joint (for joint 2) and the other ones are linear, eventually with more SSVs for each joint (fig. 5.1c).

- ◇ aut: it is the automaton used for collision detection in hybrid algorithm. The number refers to the same enumeration as described is Section 3.4.1;
- ◇ thr: it is the threshold used to split the trajectory into sub-intervals, employed to find robot sweeps;
- ◇ time: it is the total time required to perform algorithm.

As can be seen, we summarized only the main results, so not all possible combinations are showed for every configuration. Eventually we made the last tests varying only one parameter, which switches between different SSV models.

Results overview

The first thing to notice is that, except for two cases, the processing time limit is almost always met, which is the main result we wanted to obtain.

The first group of simulations (labelled with "A") is the best case for the overall algorithm and it represents two robots working behind each other, for which collision absence is evaluated at first step, comparing entire robot sweeps. Incidentally, this was one of the worst cases for the unoptimized algorithm, which conversely now is coherent. Since there is no more detailed abstraction level involved, it does not make sense to vary the number of points or SSV model, as it would produce the same results. The only critical parameter here is the threshold for robot sweeps, as eventually we are going to check collision between each sector combination, totalling $s_1 s_2$ checks, where s_1 and s_2 are the number of intervals for which the trajectory is split. Therefore it is not surprising that processing time is rising up quickly when reducing threshold value.

The second group of simulations is the one described in the previous section. First of all we tried to double the number of points for each path, with a consequent rising time, then we reduced the sector angle, obtaining similar results to the previous one. Here then we switched to a different SSV model to test whether a more complex SSV model takes its own benefits. As a result, we could not see any relevant difference between these alternative models and the basic one. In fact, on one side there is less probability to need a more detailed model (e. g., 18-DOP model), but on the other hand the processing time is a bit raised. We then tried a different automaton among the four described in Section. 3.4.1. Here again we couldn't find any noticeable difference.

The reason why these parameters do not affect significantly the final execution time is because they act on the unoptimized algorithm, which the final algorithm is trying to avoid to invoke in order to save time when there is no collision. If we do not consider the optimization, the results would be significantly dependent on these parameters. Anyway, since it is the most complete and sometimes the processing time is slightly faster, we chose to keep automaton 3 as reference for the next simulations. We finally made a test with automaton 2, which skips SSV modelling for each joint.

Unlike what can be expected, the final processing time is not very far from the other ones. This is because collision detection phase between joint blocks is always performed before getting to hybrid algorithm. However, this is still not convenient compared to the other automata.

Group "C" of simulations is similar to the previous one, where we dropped the less significant results. The final outcomes follow the same explanations of correspondent simulations of group "B".

Group "D" represents two colliding trajectories. Parameters are chosen according to group "B". We can see how detection times in this case are significantly lower compared to collision-free paths, since there is no need to check the whole trajectories. In particular, as there is a true collision and it has to be detected at the most precise level available according to the automaton chosen, there are some differences between the automata. As there can be some false alarms close to the collision point, level 2 present in automaton 3 allows to reach the goal in less time, which is one reason why we chose this as default automaton. A last simulation is performed with automaton 4, which skips all Box modelling. This is clearly faster, but can lead to poorer results. However, the time gain is so low that it does not make sense to apply this on the final algorithm.

In last simulation groups we only varied the SSV model. Group "F", in particular, is quite similar to group "B", where robot sweeps are very close to each other, but eventually there is a collision. All false alarms that may occur then are rising computation time, keeping it within the limits anyhow.

Some of the results on Table 2 were taken in order to just show the behaviour when some parameters are wrongly set. This is exactly the case where the two simulations exceed the limit. In fact, while all simulations are performed with about 50 knots for each trajectory, in one case this number is almost doubled, while in the second case the trajectory is split in too many intervals. Despite these simulations are out of normality, both have reported some good results.

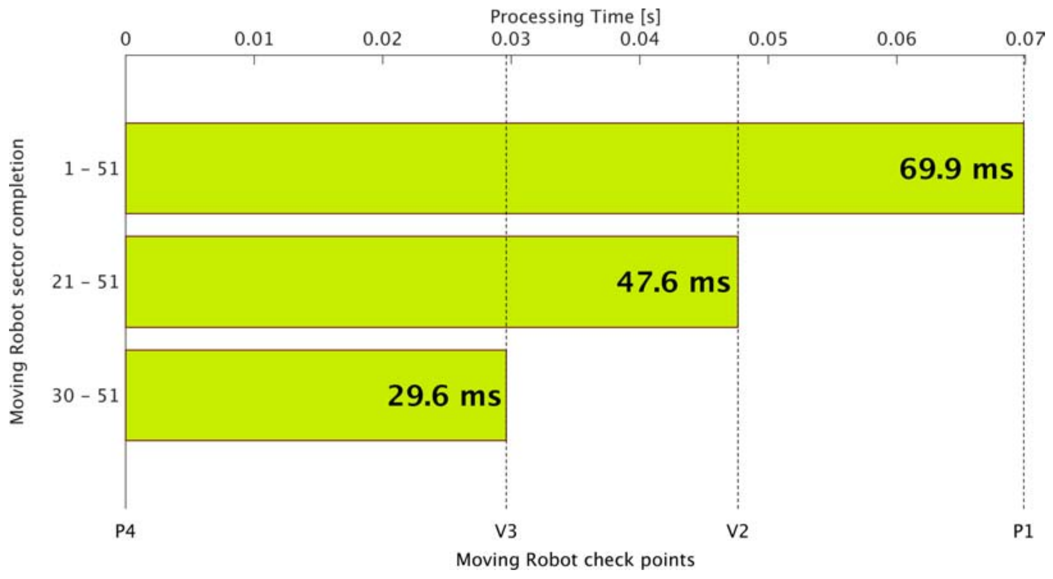
Consequences when knots are doubled

If we consider the first one, the time required for processing is nearly doubled. However, if we consider the behaviour of simple collision detection algorithm without optimization, this should have been 4 times longer, as the number of point pairs is four times greater. Eventually all the benefits must have come from the algorithm optimization. This is an important result, as it allows to relax the condition for which we need as few knots as possible for the final trajectory. This behaviour is not seen only in this occasion, but every time we double the number of points.

Consequences when trajectories are densely split

The other simulation hides another important result. In fact we have always considered the worst case condition, that is, we suppose to check the entire trajectory of the *Steady Robot* (sometimes called *Slave Robot*, which is the robot that is querying whether its trajectory is collision-free) with the entire trajectory of the *Moving Robot* (or *Master Robot*, which is moving and cannot be halted). But this is not always the case, as the Master Robot might be at the end of its path and so it is not necessary to consider its entire path but only its remaining one. Since both paths are split into sub-intervals, the best thing we can do is to check MR sweep sectors reversely. The more are the sub-intervals, the more are the check points which allow to eventually break the processing phase according to MR position. So if there are many intervals, in the best case the collision detection can be even faster than the best case with fewer intervals. This is shown in fig. 5.6.

Therefore, if we can know the current position of Master Robot when we begin collision detection phase, we can achieve the final result in less time.



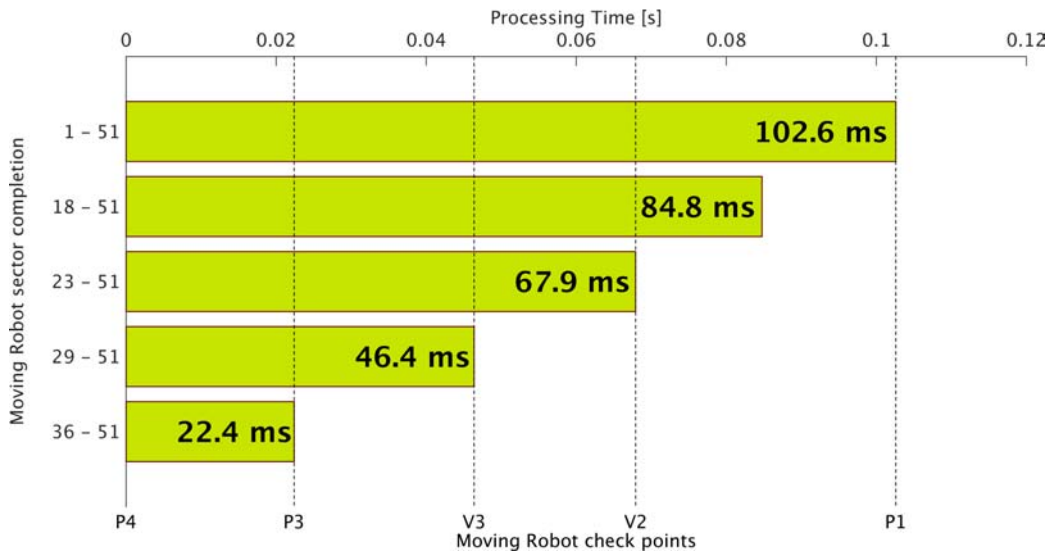


Figure 5.6: Collision-free trajectory timings. On the top, sector threshold was set to 140°, whereas on the bottom it was set to 90°. Horizontal axis with top scale represents the processing time required to detect collision absence. Since we are checking the whole path of Slave Robot with sub-paths of Master Robot in temporal-reverse order, the bottom scale represents the points along the MR trajectory over which there is certainly no collision, where always P_i, $i = 1, \ldots, 4$ are target points of `Jump3` motions and V_i are via points, enumerated in chronological order. This means that after the time indicated inside every bar we are sure that Master Robot sub-trajectory made of the set of knots indicated on the vertical axis is not colliding with the Slave Robot. The last via point which is inside this set is reported on the bottom axis. This is why we do not encounter all points on the MR path. These graphs show how, despite taking longer in the worst case, the second subdivision allows to reach the goal in less time on the best case, which can happen when SR is querying for collision when MR has almost completed his trajectory.

6 Conclusions

The aim of this work was to develop a system to detect collisions between two robot manipulators. We developed a first collision algorithm which validity is totally general, and then we optimized it by making it real-time executable, focusing on anthropomorphous robots. This algorithm needs to know real robot trajectories, which are estimated according to practical smoothing made by the controller in our case study. Despite having focused on Epson® manipulators, this behaviour is assumed by many manipulators, where additional informations might be available, making the estimation process more reliable and easier to perform.

Tests to sample paths have been performed for collision detection phase, including critical situations which must be handled within target time. During the development phase many parameters have been made available, which effect has been compared during simulations. The algorithm developed allowed to combine both reliability of Box modelling and fastness of SSV models, behaving a notable improvement to the first one, without any loss in reliability.

The optimized algorithm allowed then to reach the goal within 0.1 s for all simulations with correct parameters tuning and in worst case conditions. Eventual colliding trajectories can be detected even in less time, thanks to the algorithm structure.

This system then has been integrated into previous work, in order to be put into production in a real environment. Despite this, there has not been any practical test on collision detection phase, because the estimation phase has been carried out in another thesis work. However, simulations employed parameters which are compatible with real robot trajectories. One of these was the number of knots for the entire trajectory, which can affect significantly processing time for collision detection. Optimized algorithm, by the way, allowed to slightly relax this condition.

A user interface has been developed in order to build an SSV model for each joint of the manipulator. With this GUI the user can see the real time behaviour of parameters imposed for each link, which can be different for each one. The user can then choose the type of SSV to employ, which can affect processing time. However, after some tests, no significant differences have been found between simple and complex models, as a result of optimization done which is at a higher abstraction level and aims to avoid unnecessary steps, including the ones which involve SSV joint description.

Despite having been largely optimized, some future improvements can be performed. One could be to provide a more general method which considers any type of manipulator or any possible robot configuration inside the workcell.

Another phase which can be the natural prosecution of this thesis is following collision avoidance stage, which aims to design a new trajectory that avoids all obstacles inside the workcell, including another robot, indeed.

Finally, all simulations and studies are carried out with a single movement, where *Master* and *Slave* Robot roles are fixed. Once a movement is terminated, by the way, robot roles must be swapped, so we can iterate the algorithm continuously.

A

Trapezoidal-speed law relationships

If we want to drive one motor from a certain position $\mathbf{q}_0$ to a target position q_f there are different motion laws that can be planned. One of the simplest one is the trapezoidal speed law, which ensures continuity both on position and speed (fig. A.1).

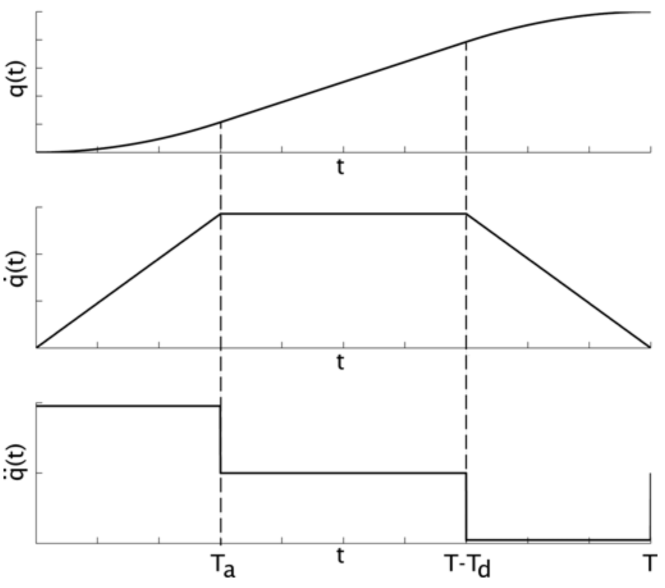


Figure A.1: Trapezoidal speed law time trend for position, speed and acceleration.

The drive law can be split in 3 parts, called *acceleration*, *cruise* and *deceleration*. The middle stage is covered at constant speed (V_{cr}), while in the other two stages the acceleration is constant (respectively a and d). The overall law time is called *drive time* (T), while T_a means time required to make acceleration phase, T_{cr} is the time driven at constant speed and finally deceleration is performed in T_d.

As the speed law is described as a piecewise-linear function (from which it inherits the name), it is easy to obtain dynamic relationships depicted in fig. A.1:

$$q(t) = \begin{cases} q_0 + \frac{1}{2} V_{cr} \frac{t^2}{T_a} & 0 \leqslant t \leqslant T_a \\ q_0 + V_{cr} \left(T - \frac{T_a}{2} \right) & T_a < t \leqslant T - T_d \\ q_f - \frac{1}{2} V_{cr} \frac{(T-t)^2}{T_d} & T - T_d < t \leqslant T \end{cases} \tag{A.1}$$

$$\dot{q}(t) = \begin{cases} V_{cr}\frac{t}{T_a} & 0 \leqslant t \leqslant T_a \\ V_{cr} & T_a < t \leqslant T - T_d \\ V_{cr}\frac{T-t}{T_d} & T - T_d < t \leqslant T \end{cases} \tag{A.2}$$

$$\ddot{q}(t) = \begin{cases} \frac{V_{cr}}{T_a} & 0 \leqslant t \leqslant T_a \\ 0 & T_a < t \leqslant T - T_d \\ -\frac{V_{cr}}{T_d} & T - T_d < t \leqslant T \end{cases} \tag{A.3}$$

Depending on input parameters, next relationship may be useful:

$$V_{cr} = \frac{q_f - q_0}{T - \dfrac{T_a + T_d}{2}} \tag{A.4}$$

If we suppose the law to be symmetric (i. e., $T_a = T_d$ and $a = d$) we can simplify the previous relationships, obtaining:

$$q(t) = \begin{cases} q_0 + \frac{1}{2}V_{cr}\frac{t^2}{T_a} & 0 \leqslant t \leqslant T_a \\ q_0 + V_{cr}\left(T - \frac{T_a}{2}\right) & T_a < t \leqslant T - T_a \\ q_f - \frac{1}{2}V_{cr}\frac{(T-t)^2}{T_a} & T - T_a < t \leqslant T \end{cases} \tag{A.5}$$

$$\dot{q}(t) = \begin{cases} V_{cr}\frac{t}{T_a} & 0 \leqslant t \leqslant T_a \\ V_{cr} & T_a < t \leqslant T - T_a \\ V_{cr}\frac{T-t}{T_a} & T - T_a < t \leqslant T \end{cases} \tag{A.6}$$

$$\ddot{q}(t) = \begin{cases} \frac{V_{cr}}{T_a} & 0 \leqslant t \leqslant T_a \\ 0 & T_a < t \leqslant T - T_a \\ -\frac{V_{cr}}{T_a} & T - T_a < t \leqslant T \end{cases} \tag{A.7}$$

while (A.8) becomes now

$$V_{cr} = \frac{q_f - q_0}{T - T_a} \tag{A.8}$$

There are many criteria for searching an optimal set of parameters, but in this work we do not have to design a motion law. For details about motion planning and alternative motion laws, see [12].

B

Properties of transformation matrices

With transformation matrix in this work we mean a special matrix which combines a rotation matrix $\mathbf{R}$ and an offset vector $\mathbf{t}$. We always deal with points in euclidean space when we want to apply these transforms, so we tacitly assume that $\mathbf{R} \in \mathbb{R}^{3\times 3}$ and $\mathbf{t} \in \mathbb{R}^{3\times 1}$. To be more precise, $\mathbf{R}$ belongs to a group called $SO(3)$ and it has some important properties, which main one is that it is orthogonal, so

$$\mathbf{R}^{-1} = \mathbf{R}^{T}$$

This transformation is used whenever we want to denote a certain point (which is supposed to be referred to a specific *reference frame*) in another reference frame.

The combination of rotation and translation defines a *rigid transformation*, which is an *affine transformation*:

$$\mathbf{q} = \mathbf{R}\mathbf{p} + \mathbf{t} \tag{B.1}$$

where $\mathbf{p} \in \mathbb{R}^{3\times 1}$ is the point we want to transform and $\mathbf{q} \in \mathbb{R}^{3\times 1}$ is the transformed point. To compact the notation, rotation and offset are embedded into the transformation matrix:

$$\mathbf{T} = \left[\begin{array}{c|c} \mathbf{R} & \mathbf{t} \\ \hline 0 & 1 \end{array}\right] \quad \text{so that} \quad \left[\begin{array}{c} \mathbf{q} \\ \hline 1 \end{array}\right] = \left[\begin{array}{c|c} \mathbf{R} & \mathbf{t} \\ \hline 0 & 1 \end{array}\right] \left[\begin{array}{c} \mathbf{p} \\ \hline 1 \end{array}\right]$$

This transformation belongs to another special group called $SE(3)$.

Thanks to the orthogonality property of $\mathbf{R}$, if we want to revert the relation (B.1), after few trivial steps we find that

$$\mathbf{p} = \mathbf{R}^{-1}(\mathbf{q} - \mathbf{t}) = \mathbf{R}^{T}(\mathbf{q} - \mathbf{t}) \tag{B.2}$$

which can be resumed into the inverse transformation matrix

$$\mathbf{T}^{-1} = \left[\begin{array}{c|c} \mathbf{R}^{T} & -\mathbf{R}^{T}\mathbf{t} \\ \hline 0 & 1 \end{array}\right] \quad \text{so that} \quad \left[\begin{array}{c} \mathbf{p} \\ \hline 1 \end{array}\right] = \left[\begin{array}{c|c} \mathbf{R}^{T} & -\mathbf{R}^{T}\mathbf{t} \\ \hline 0 & 1 \end{array}\right] \left[\begin{array}{c} \mathbf{q} \\ \hline 1 \end{array}\right]$$

In this way it is easy to see that reversing a rigid transformation can be very quick compared to the usual inverse matrix procedure. For further details about rotation matrices, see [4].

Rotation Matrices evaluation

A typical situation where a rotation matrix is employed is when we want to obtain a new reference frame by rotating another frame around a certain axis by a certain angle. Therefore we can have 3 basic situations, one for each main axis:

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix} \tag{B.3}$$

$$\mathbf{R}_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix} \tag{B.4}$$

$$\mathbf{R}_z(\gamma) = \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{B.5}$$

If we want to combine consecutive rotations, we will eventually refer to these primitives, obtaining another matrix which still belongs to $SO(3)$ and then inherits all properties explained above.

C

Kinematic model of robot manipulators

Reference frames

In a typical workcell it is usual to define many reference frames, in order to simplify points localization and their processing (fig. C.1).

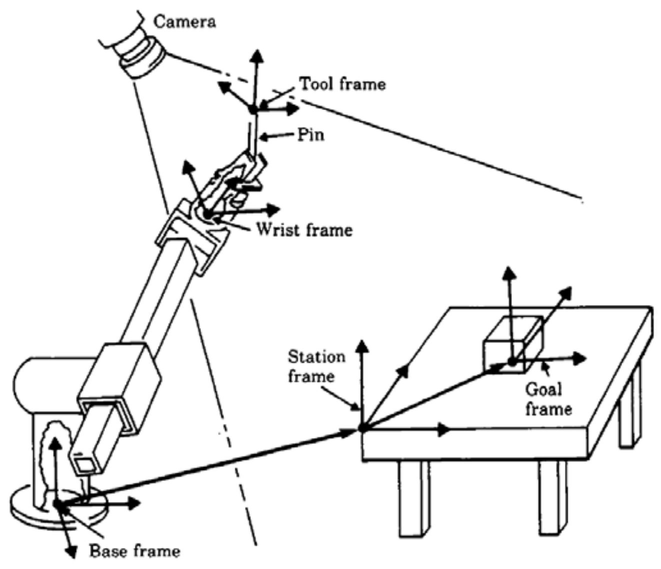


Figure C.1: Common defined reference frames in a workcell (source: [4]).

Most common reference frames are:

- *Station Frame* (S): this identifies the origin of the workcell and main axes. All equipment inside the workcell that has to cooperate with another device uses this frame as an interface, as this is common between all devices;
- *Base Frame* (B): also known as *Robot Frame*, it is defined for each manipulator in the workcell. This frame is defined relative to the Station Frame;
- *Wrist Frame* (W): it is attached to the last link of the robot. Its relative definition to the Base Frame depends on the current robot joint configuration;
- *Tool Frame* (T): it is located on the end-effector, therefore its definition relative to the Wrist Frame is always static, unless we are changing the tool.
- *Goal Frame* (G): it is attached to the object we want to pick up, so it is usually defined relative to the Station Frame.

Joint Space vs Cartesian Space

If we want to refer to a specific point within the workspace, we can express it in two different ways:

- *Joint Space*: this is an n-dimensional space, where n is the number of robot links. In this notation, coordinates are positions of each motor.
- *Cartesian Space*: this is a 6-dimensional space and each vector belonging to this space is made by 3 position and 3 orientation coordinates, due to the 6 degrees of freedom each rigid object has.

Differences and motivations between these notations can be found in every robotics book such as [4] and [12]. The important thing is that mapping from Joint to Cartesian space is always well defined, while to perform reverse transform we need to know which configuration is used. In fact, given a point in Cartesian space, it can be reached with different joint configurations (e. g., above or below elbow). If we suppose to know which configuration is used, the reverse mapping can be performed obtaining a unique result.

The mapping between Joint space and Cartesian space is obtained with the *direct kinematics* (also called *forward kinematics*), while the reverse one is called *inverse kinematics*. For more details about these aspects, see [4] [12].

Denavit-Hartenberg parameters

If we want to perform a certain movement and pick up a target object, we must let Tool and Goal frame coincide. To do this, we have to know the transformation matrix between the end-effector and the station frame, as long as we suppose to know transformation between goal and station frame.

Let $\mathbf{T}_{TS}$ be the transformation matrix we want to evaluate. This can be split into consecutive transformations:

$$\mathbf{T}_{TS} = \mathbf{T}_{BS} \cdot \mathbf{T}_{WB} \cdot \mathbf{T}_{TW} \tag{C.1}$$

The only non-static matrix in (C.1) is $\mathbf{T}_{WB}$, as the robot base is static and the tool is supposed to be always the same.

Let $\mathbf{T}_{i,i-1}$ be the transformation matrix which maps a point known on the reference frame attached to joint i to the reference frame on joint $i-1$. The problem is then postponed to find the matrices which make the sequence of transformations[1]

$$\mathbf{T}_{WB} = \mathbf{T}_{1,0} \cdots \mathbf{T}_{n-1,n-2} \cdot \mathbf{T}_{n,n-1} \tag{C.2}$$

Eventually, we have to compute joint transformation matrices $\mathbf{T}_{i-1,i}$. We always will be dealing with 1 degree of freedom joints, so the unknown in this problem is just one. If the joint is *prismatic*, the variable is an offset, whereas if it is *rotational* it is an angle. To compute this matrix we use the universal *Denavit-Hartenberg* notation,

1 here wrist frame is also indicated with n, as the number of robot joints, while 0 frame is robot base

which allows to resume each transformation with 4 parameters, which are two angles and two offsets (fig. C.2):

- ◇ a_{i-1} is the distance from $\hat{Z}_{i-1}$ to $\hat{Z}_i$ measured along $\hat{X}_{i-1}$;
- ◇ α_{i-1} is the angle between $\hat{Z}_{i-1}$ and $\hat{Z}_i$ measured about $\hat{X}_{i-1}$;
- ◇ d_i is the distance from $\hat{X}_{i-1}$ to $\hat{X}_i$ measured along $\hat{Z}_i$;
- ◇ θ_i is the angle between $\hat{X}_{i-1}$ and $\hat{X}_i$ measured about $\hat{Z}_i$.

For more details about the definition of reference systems employed in these parameters, see [4].

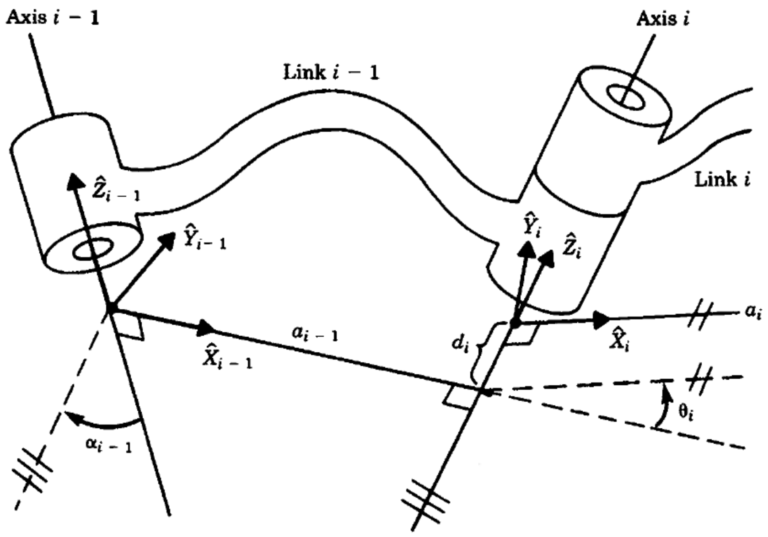


Figure C.2: Denavit-Hartenberg parameters (source: [4]).

Transformation matrix is then obtained as

$$\mathbf{T}_{i,i-1} = \mathbf{T}_{R_x}(\alpha_{i-1}) \cdot \mathbf{T}_{S_x}(a_{i-1}) \cdot \mathbf{T}_{R_z}(\theta_i) \cdot \mathbf{T}_{S_z}(d_i) \tag{C.3}$$

where

$$\mathbf{T}_{R_z}(\theta_i) = \left[\begin{array}{c|c} \mathbf{R}_z(\theta_i) & 0 \\ \hline 0 & 1 \end{array}\right] \quad \text{and} \quad \mathbf{T}_{S_z}(d_i) = \left[\begin{array}{c|c} & 0 \\ 0 & 0 \\ & d_i \\ \hline 0 & 1 \end{array}\right]$$

where $\mathbf{R}_z(\theta_i)$ is defined in Appendix B, with obvious extension for x axis. The final result is

$$\mathbf{T}_{i,i-1} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_{i-1} \\ \sin\theta_i \cos\alpha_{i-1} & \cos\theta_i \cos\alpha_{i-1} & -\sin\alpha_{i-1} & -d_i \sin\alpha_{i-1} \\ \sin\theta_i \sin\alpha_{i-1} & \cos\theta_i \sin\alpha_{i-1} & -\cos\alpha_{i-1} & d_i \cos\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{C.4}$$

An important observation is that to compute (C.2) we always need to know previous link transformation, so this procedure must be iterated serially from link 1 to link n. Incidentally, if we halt this product at a certain link m (i. e., we have already computed $\mathbf{T}_{m-1,0}$), we can know the position of DH frame origin of link m. In fact in m-th link reference frame this has coordinates $(0, 0, 0)$. Therefore, the origin of joint m referred to the base frame can be evaluated as

$$^{B}\mathbf{O}_m = \mathbf{T}_{m,0} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \mathbf{T}^{(4)}_{m-1,0} \tag{C.5}$$

where the superscript means the 4-th column of $\mathbf{T}_{m-1,0}$.

In this work we refer to *kinematic model* as the set of parameters and matrices which allow to describe robot arm orientation and position given a certain joint configuration. The central entities in this manner are transformation matrices $\mathbf{T}_{i,i-1}$, which are the only ones to need to be updated constantly.

Bibliography

[1] Ahmad Yasser Afaghani and Yasumichi Aiyama. On-line collision avoidance of two command-based industrial robotic arms using advanced collision map. *Journal of Robotics and Mechatronics*, 2014.

[2] Paul Bosscher and Daniel Hedman. Real-time collision avoidance algorithm for robotic manipulators. *Industrial Robot: An International Journal*, 38(2):186–197, 2011.

[3] Oliver Brock and Oussama Khatib. Mobile manipulation: Collision-free path modification and motion coordination. In *Proceedings of the 2nd International Conference on Computational Engineering in Systems Applications*, pages 839–845, 1998.

[4] J.J. Craig. *Introduction to Robotics: Mechanics and Control*. Addison-Wesley series in electrical and computer engineering: Control engineering. Addison-Wesley, 1989.

[5] David Eberly. Robust computation of distance between line segments. Technical report, Geometric Tools, LLC, 2014.

[6] Kai Hormann and Alexander Agathos. The point in polygon problem for arbitrary polygons. *Comput. Geom. Theory Appl.*, 20(3):131–144, 2001.

[7] Ji-Hun Kim Jung-Jun Park and Jae-Bok Song. Path planning for a robot manipulator based on probabilistic roadmap and reinforcement learning. *International Journal of Control, Automation, and Systems*, 2007.

[8] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *Robotics and Automation. Proceedings. 1985 IEEE International Conference on*, volume 2, pages 500–505, Mar 1985.

[9] B.H. Lee and C.S.G. Lee. Collision-free motion planning of two robots. *Systems, Man and Cybernetics, IEEE Transactions on*, 17(1):21–32, Jan 1987.

[10] Riccardo Muraro. Modellizzazione tridimensionale di celle robotizzate flessibili. Master's thesis, Universitá degli Studi di Padova, 2014.

[11] P.A. O'Donnell and T. Lozano-Periz. Deadlock-free and collision-free coordination of two robot manipulators. In *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, pages 484–489 vol.1, May 1989.

[12] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics: Modelling, Planning and Control*. Advanced Textbooks in Control and Signal Processing. Springer, 2009.