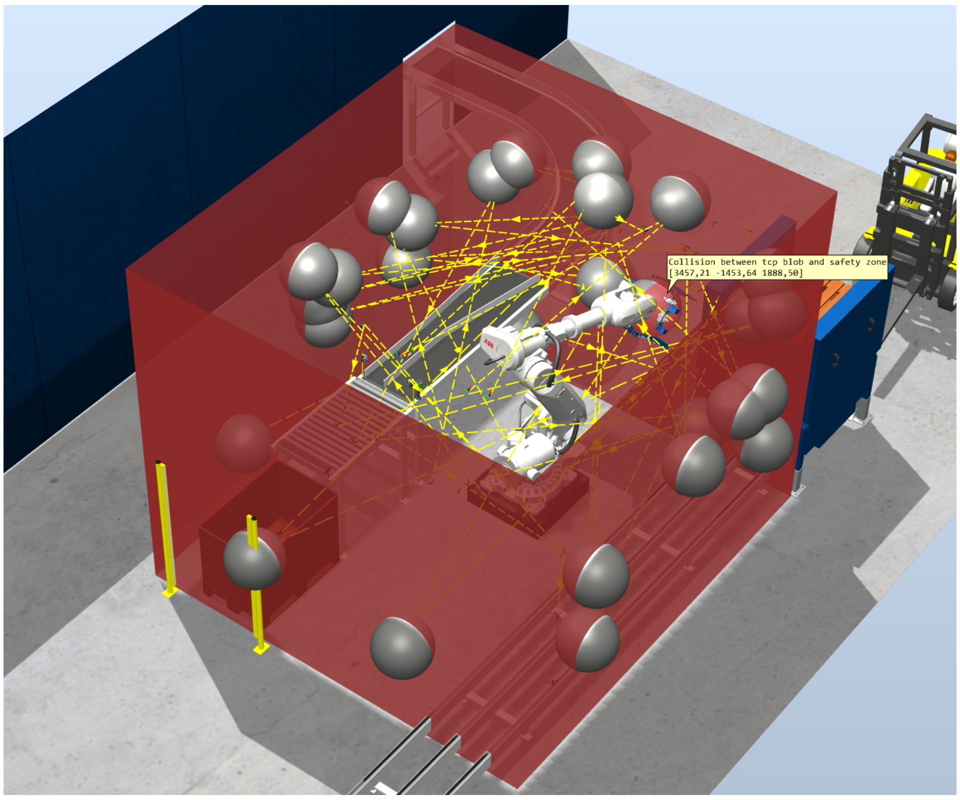
Collision between tcp blob and safety zone
[3457,21 -1453,64 1888,50]

Defining and minimizing cell margins in a SafeMove supervised robot system

The development of an algorithm that assists the robot cell engineer at the design phase in RobotStudio

Master's thesis in Systems, Control and Mechatronics

Oskar Henriksson

EX066/2018

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Master's thesis EX066/2018

Defining and minimizing cell margins in a SafeMove supervised robot system

The development of an algorithm that assists the robot cell engineer at the design phase in RobotStudio

Oskar Henriksson

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2018

Defining and minimizing cell margins in a SafeMove supervised robot system
The development of an algorithm that assists the robot cell engineer at the design phase in RobotStudio
Oskar Henriksson

Supervisor: Henrik Berlin, ABB Robotics
Examiner: Martin Fabian, Signals and Systems

Master's Thesis EX066/2018
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by [Name of printing company]
Gothenburg, Sweden 2018

Defining and minimizing cell margins in a SafeMove supervised robot system
The development of an algorithm that assists the robot cell engineer at the design phase in RobotStudio

Oskar Henriksson
Department of Electrical Engineering
Chalmers University of Technology

Abstract

When designing a robot cell, it is often desired to keep it as small as possible, while at the same time meeting a required cycle-time. A tight cell increases the risks of costly fence-robot collisions, to avoid this the velocity and boundary supervision system SafeMove can be used. SafeMove halts the robot upon a zone or velocity violation, though the stop is not atomic, a brake distance has to be accounted for. To approximate these brake distances at the design phase, where changes are cheap, the cell engineer only has standard brake scenarios at help. To ease the process and minimize the total brake margin between the cell boundary and fence, this thesis proposes a two-step algorithm. First, it explores and describes the relationship between the velocity-programmed trajectory, cycle-time, and brake margin to the engineer using RobotStudio. Secondly, by mixed integer linear programming, it minimizes the brake margin by adjusting the programmed trajectory. The algorithm successfully describes and visualizes the relationship between cycle-time and occupied factory floor area for a given trajectory and a set of velocities. Also, the optimizer, with the objective to minimize maximum angular velocity, lowers the maximum velocity for the tested trajectory and robot.

Keywords: Brake margin, fence, trajectory optimization, MILP, SafeMove, RobotStudio

Acknowledgements

Firstly, I want to thank you, Lisa, you are always loving, always supportive, always understanding and always remembering me about the great life beyond the computer screen. Gratitude to my family and friends who never tried to change who I am and always putting trust in me when my own belief was lacking.

I am deeply grateful for all the support, discussions, knowledge and positive energy Henrik Berlin as a supervisor at ABB has contributed to. Thank you, Niklas Skoglund at ABB for the help with the RobotStudio API and all the interesting visionary talks. Thank you, August Ramle at ABB for being the archetype of a experimenting, inspiring, positive engineer. Thank you, Hans Andersson at ABB, providing the modified virtual controller with live stopping distances and your expertise about emergency stops. Also, to all of you great colleagues at ABB Robotics Mölndal a great thank you for being such a social, creative and supportive group of people.

Thank you, Martin Fabian for examining this master's thesis, putting me in the right direction and supporting with all the practicalities. Finally, thanks to all great people contributing to the open source community and the publicly available libraries, programming languages and tools, reinventions of the wheel could be avoided.

Oskar Henriksson, Gothenburg, June 2018

Contents

List of Figures

1
Introduction

In industrial robot cells today, fences protect not only humans from entering the cell, but also prevent the robot from moving outside of it. Cells are kept as small as possible, often resulting in a robot working range that is greater than what the cell allows. The fence does not only have to be strong enough to withstand the robot's full force [1], but is also expensive in purchase and installation. If an incident occurs, the fence most certainly needs to be replaced and maybe also parts of the robot and its tool. Smaller robot cells are a consequence of the desire to fit as many production units as possible into a limited floor area.

There exists a supervising system called SafeMove (explicitly available for ABB Robots), which allows the cell constructor to set up virtual zones for the robot. SafeMove halts the robot if it violates the virtual boundary of the zone, hence, the purpose of the fence, if placed correctly, is reduced to prevent humans from entering the cell. However, halting a moving robot results in a significant brake distance [2], thus, a brake margin in-between the real fence of the cell and the virtual boundary must be accounted for. This brake margin depends on the velocity of the robot [3], therefore SafeMove includes a velocity restriction, which if violated, halts the robot even though it is within the virtual boundary.

1.1 The problem

The SafeMove setup procedure starts in the virtual robot simulation software Robot-Studio, where the virtual zones, stopping category and velocity supervision can be configured. This is typically done after the virtual robot and the cell has been programmed. Once the SafeMove configuration has been set, as part of the safety standard [1], a unique hash is generated from the configuration to prevent unnoticed changes.

When the cell and robot has been physically installed, the safety configuration with its unique hash is tested and signed by the responsible on site. The test procedure focuses on manually programmed trajectories that purposely violates the safety configuration, to confirm that SafeMove stops the robot before a collision occurs. If the configuration is unsatisfying being too strict or too loose, changes are expensive, both in time and money.

To prevent unsatisfying safety configurations brake margins are manually approximated by studying scenario-based braking distances for the particular robot defined by *ISO 10218-1* [2]. These scenarios are very much simplified, hence it is hard for a cell constructor to apply the scenarios to an ongoing cell design. And there is no possibility to simulate this in RobotStudio today. These scenario-based stops has been verified and repeated by [3] for some robots.

1.2 The solution

To assist the cell configurator in an early stage, this thesis proposes a two-phase algorithm, which presents and reduces the dead floor area consumed by the zone-fence margin. Firstly, the robot cell and the programmed robot trajectory is analyzed such that an approximated fence margin is achieved. Secondly the zone-fence margin is reduced by an optimization algorithm by lowering the top-velocity in the programmed robot trajectory. Finally, the fence position is determined by the zone-fence margin.

The first phase of the algorithm utilizes *RobotStudio*, a virtual robot simulation tool to simulate ABB Robots. Since the cell and its safety configuration is created in RobotStudio, both simulations and measurements can be done without access to a real robot. The algorithm has a generic approach, and with the purpose to be applicable to any RobotStudio cell.

Obtained explicitly for this thesis, a virtual robot has been modified by the motion team at ABB Robotics to produce data about the exact stop position and time if an emergency stop where issued. The stopping data is computed in each discrete time step, hence it is an approximation based on the internal dynamic model of the robot. Without the modified virtual robot, an alternative dynamic estimation model would be required to predict brake distances.

2
Theory

An industrial robot consists of two parts, a *manipulator* and a *controller*. The manipulator itself is a dumb, mechanical component, meanwhile the often-separate control system referred to as the *controller* houses a real-time computer, an electrical control system and often a programmable logic computer(PLC). The supervising system SafeMove is also located in the controller, utilizing a dedicated so-called safety PLC to communicate with surrounding industrial components in the factory such as light guards and emergency-stop buttons.

Since this thesis only focuses on simulations, the focus is put on the manipulator and its kinematics, rigid body transformations in the cartesian coordinate system and optimization. As the focus is on the proposed algorithm, the only virtual robot used in this thesis is the IRB 6700-175/3.05 from ABB, where the manipulator has six degrees of freedom(6-DOF).

2.1 Anatomy of the 6-DOF decoupled manipulator

The generic six-axis decoupled manipulator used in this thesis features three positional and three orientational joints. Hence, the inverse kinematic problem (described in section 2.6) of finding joint positions that places the TCP at a certain point in joint space can be *decoupled* into a positional and an orientational problem, thus, be solved independently. On the IRB 6700-175/3.05, shown in Figure 2.1, axes 1, 2 and 3 are positional and axes 4, 5 and 6 are orientational. The *decoupling* requires the three orientational joints to rotate around a common origin, creating a spherical joint referred to as the *wrist center point (WCP).*

2.1.1 Wrist center point (WCP)

The Wrist Center Point (WCP) is defined by the point where the rotational axes of joints 4, 5 and 6 intersects, this is discussed and illustrated further in section 2.7. The joints are situated in a cascading manner, thus, the end position of axis 6 depends on axis 5 which depends on axis 4.

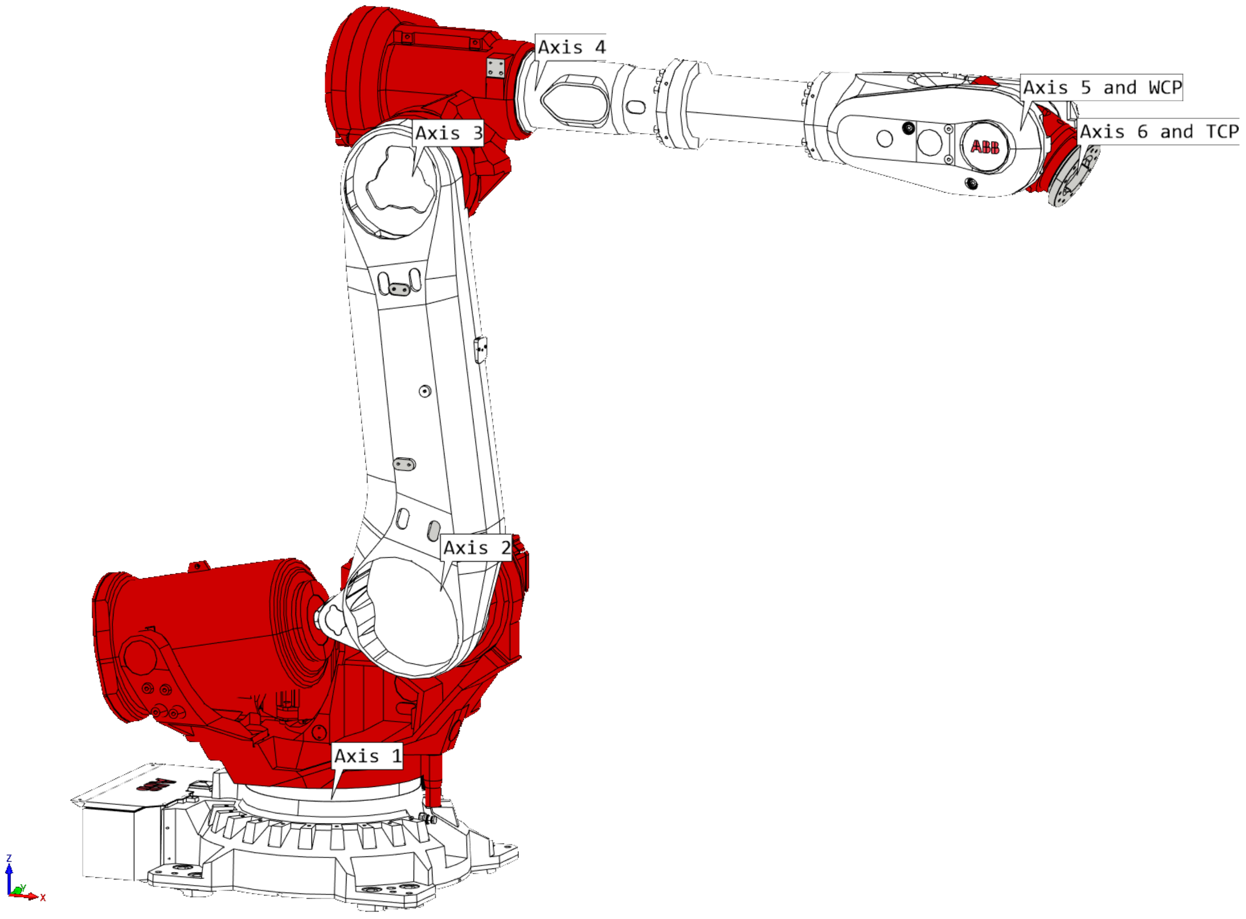

Figure 2.1: Anatomy and joint positions of the IRB 6700-175/3.05, where the contrast colored parts are moved by axes 1,3 and 5 and the white colored parts are moved by axis 2, 4 and 6.

2.1.2 Tool center point (TCP)

The tip of the manipulator, referred to as the Tool Center Point (TCP), is a point positioned with a rotational and a translational offset from the WCP, which can be described by a frame, described in subsection 2.1.3. Sometimes, the TCP are subject to change when attaching equipment (a tool) to the robot.
There is still a need for new word abbreviation though, if a tool is attached, the tool's defined as the *end effector(EE)*, which is an offset from the TCP to a point on the attached equipment.

2.1.3 Frame

A frame is simply a local coordinate system with a certain offset and rotation to its parent frame, the mathematical definition is found in subsection 2.5.3. Frames are used to ease the programming process, introducing modularity, which eases future changes in the program, but also to describe the actual kinematics of the robot, discussed and illustrated in section 2.7. From a ABB point of view (including

RobotStudio), there are a couple of standard frames, forming a hierarchical structure illustrated in Figure 2.2, where the *world frame* is the world origin. Additionally, there are three more frames, the *task frame*, a direct ancestor to the *world frame* and parent to all other frames, the *manipulator base frame* locates the base of the manipulator(if multiple, there are multiple base frames). The manipulator base frame is naturally a direct ancestor to the *controller world frame*, which has one or multiple children named *object frame(s)* that houses the programmatic positions and trajectories to be known by the robot.

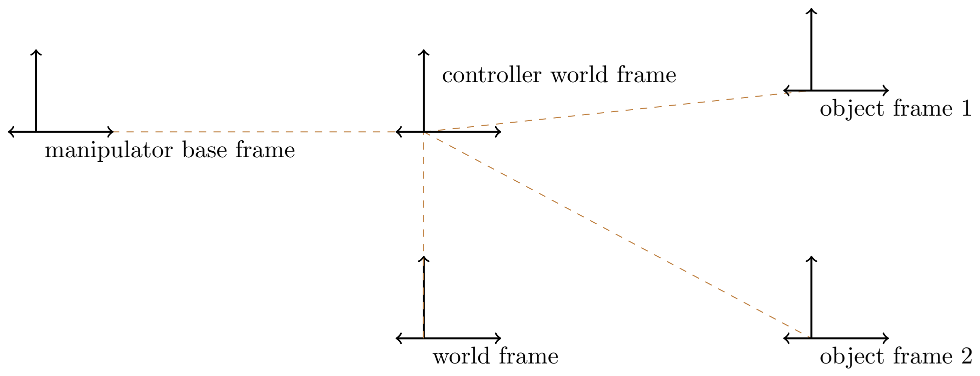

Figure 2.2: The standard programmatic frames of ABB robots where the robot is positioned in the local origin of the base frame, and all programmatic targets in the work frame.

2.2 Robot motion programming

A robot motion can be described, defined and programmed in various ways, from low-level absolute joint positions to more application based *targets* (a frame without children), with trajectories in-between. Low-level trajectories are usually described as a sequence of joint sets discretized with a certain resolution. Each joint set is referred to as a pose, as proposed by [4]. A motion can then be discretely defined by a set of poses with a given time in-between each pose.

2.2.1 high-level motion programming of ABB robots

Most robots also offer more high-level programming. In the case of ABB robots, trajectories can be composed by *targets*, which specifies The TCP position by a frame and a specified type of movement in-between, known as a *instruction* [5]. For most application based robot cells, the high-level description of the trajectory is the most practical. Unfortunately, the new description introduces other problems; target reachability and compatibility, if the target is within reach, it is often reachable from different poses. Each unique pose to a target is denoted as a *configuration*, which introduces a compatibility issue discussed more in subsection 2.2.1

Instructions

Moving between two targets can be done either in the shortest distance in cartesian space or in the shortest distance in joint space. Moving in joint space is almost always possible, but it is not suited for precision-type applications since the TCP trajectory between the two targets is unintuitive for a human. Moving in a linear fashion instead ensures a straight line in cartesian space between the two targets, but adds the requirement of compatible *configurations* between targets, to be known more about in subsection 2.2.1.

Each instruction is defined with a certain flexibility in precision, denoted as a *zone*, which is an offset radius from the target origin. The robot is guaranteed to pass through the zone, but not through the target origin, if not explicitly specified.

Each instruction also holds a velocity parameter which defines the desired TCP velocity in mm/s towards the target zone. When the target zone has been reached, the robot starts to execute the next instruction according to its properties. On ABB robots, this is done live at each execution, thus, the robot does not foresee further in the program than one step at the time. This is by design since it makes the movements repeatable and reliable, independent on instructions further ahead.

Configurations

A 6-DOF manipulator can in some scenarios reach a target using eight different poses [6]. Similarly a human arm (with 7-DOF), can reach a object from various poses. In the case of IRB 6700-175/3.05, axis 1 can turn around and the target, which is now behind the manipulator, is still reachable. Also axes 2 and 3 can create a "mirrored" pose like any double hinge joint, lastly, the spherical joint at the WCP can achieve the same rotation with different two different poses of axes 4,5 and 6. This totals into a maximum of $2^3 = 8$ configurations if no joint limits are hit and the target is placed within reach for all of the alternatives.

When programming linear trajectories, the configuration must be specified and *compatible*, meaning that no reorientation is required between the targets. Similarly as when putting on a suit jacket, if one starts from the wrong pose with the arm it is impossible to get your arm through straight without ruining the jacket. The very same situation applies to linear motion trajectories, if the joints where continuous such that there were no joint limits, the problem would not exist.

2.3 RobotStudio

A virtual robot controller in RobotStudio is a direct software replica of a real robot controller, mimicing properties, timings and behavior. The simulated results are thereby close to robots in real life(IRL). $\pm 3\%$ cycle-time difference between a real robot cell and the virtual copy in RobotStudio is guaranteed by the Virtual Robot Controller (VRC) interface specification [7], which ABB complies to.

2.3.1 Emergency stops

As of today, it is hard to simulate emergency stops in RobotStudio, the reason is that the communication with the robot controller and its internal kinematic model is terminated as soon as an emergency stop is triggered. The controller can trigger two types of stops; *category 0* and a *category 1* stop. The first applies the physical brakes directly, meanwhile *category 1* follows the robots last instruction meanwhile braking with the motors. If the robot is still moving after one second, the physical brakes are applied [2].

The modified virtual controller enhanced with brake distance approximation at each discrete time step only provides these data for *category 0* stops, hence the algorithm proposed in item 3.1.4 will use these numbers explicitly. Hence, these values are not directly suitable for SafeMove configurations which utilizes *category 1* stops. Nevertheless, the applicability of the results is not affected by stop type, thereby the algorithm proposed is directly applicable to category 1 stops whenever data becomes available.

2.4 SafeMove

The position and velocity supervising system SafeMove is used very differently depending on application and cell type. The setup can be very generic, only restricting the cell area to reduce the need of a robot-safe fence. But there are also applications where SaveMove is tailored to fit a specific robot trajectory and with a goal to minimize the robot cell area. Typically, the SafeMove configuration is added to the virtual robot cell in RobotStudio when trajectories and positions have been set. The complete configuration is then stored in a .xml file and a unique hash string is generated out of the configuration.

SafeMove supports multi-robot setups and multiple zones, where each zone can be defined using up to 24 vertices. Zone walls are only allowed orthogonally to the common xy-plane, representing the floor [8]. The position supervision is either based on the elbow of the robot (situated between axis 3 and 4) or a tool attached to the robot. Virtual convex spherical shapes are used to conservatively approximate the geometry of the elbow and/or tool, to eases collision classification of a pose without the risk of a false negative. Also, this is related to the SafeMove code's safety classification according to standard ISO 13849-1:2015 which puts certain requirements to the code. The standard also restricts allowed complexity of the supervisory algorithms.

Each SafeMove configuration must be extensively tested before put into operation by the factory safety responsible. Every zone plane, and velocity specification must be separately verified IRL to see that SafeMove acts as intended. Thereby are complex SafeMove configuration even more time consuming to test. Also, it is important to note that any added payloads are not supervised by the safety-system, hence when testing, the heaviest payload to be used in the cell, must be attached to the robot.

2.5 Rigid body transformations

Since robots move in the cartesian space, mainly the first phase of the algorithm utilizes several rigid body definitions and transformations in kinematic calculations, collision detection, zone definitions and trajectory generation.

2.5.1 Plane

The infinite plane as defined by [9], as the unique two-dimensional surface that is perpendicular to the normal, n and passes through the point P_0, where $n = Ai + Bj + Ck$, $n \neq 0$, $P_0 = [x_0, y_0, z_0]$ and i, j and k are unit vectors in R3. The relationship between n and P_0 is achieved by the position vector $r_0 = x_0 i + y_0 j + z_0 k$. The standard form denoted by [9] is then

$$Ax + By + Cz = Ax_0 + By_0 + Cz_0, \tag{2.1}$$

which can be simplified to

$$Ax + By + Cz = D. \tag{2.2}$$

Plane offset

Translating a plane in the direction of the normal n is simple, since the orientation and thereby the normal n to the plane is unchanged. By dividing all the constants in the plane by D, the same plane, but with normalized constants are achieved

$$\frac{Ax}{D} + \frac{By}{D} + \frac{Cz}{D} = \frac{D}{D}. \tag{2.3}$$

The normalized plane can then be offset with e simply by

$$D' = D + e, \tag{2.4}$$

where D' is the new offset.

Plane - plane intersection

The intersection between two planes can be found if they are not parallel. According to [9], an intersection vector v exists such that it is orthogonal to the normals n_1 and n_2 of the intersecting planes, thus:

$$v = n_1 \times n_2, \tag{2.5}$$

where $\times$ is the cross product.

To be able to define the location of the intersection as a line equation on the form $p + v$, a point of intersection p is wanted. There are three unknowns coordinates in p and there are only two plane equations, resulting in the two equations:

$$n_1 p = -A_1 x - B_1 y - C_1 z + D_1 \tag{2.6}$$
$$n_2 p = -A_2 x - B_2 y - C_2 z + D_2. \tag{2.7}$$

According to [10], a special solution can be given if one of the unknowns are assumed, specifically in this thesis $p_z = 0$ since both planes will always cut the xy-plane, described in subsection 3.1.2. With p_z given, there are only two unknowns left, Equation 2.7 can be solved and p found.

2.5.2 Collision detection, sphere - plane

Collision detection in general is very much dependent on the type of polygon and there is a smorgasbord of algorithms available. This thesis will utilize the distance e between a sphere with radius r and a plane to determine a collision, inspired by [11]. The normal to the plane as a unit vector is defined by n, to achieve the unit vector, the plane equation needs to be normalized as done in plane offset part of Equation 2.5.1.
Now, projecting the sphere's origin s onto the plane's normal as a unit vector n and subtract the plane's offset from the origin D, gives the distance e between the plane and the sphere's origin:

$$e = n \bullet s - D. \tag{2.8}$$

If a collision has occurred, $|e| < r$, other ways the plane and the sphere does not intersect.

2.5.3 The 3D frame

A frame F is a coordinate system, with a local origin and an axis rotation, from which local coordinates and rotations can be defined as described in subsection 2.1.3. A frame can be represented as a 4 by 4 matrix, with a rotational part R, and a translational part T, describing how to convert a point from the parent frame to the local frame's coordinates. The transformation of point p_0 by frame F results in the point p_1 as follows:

$$p_1 = F p_0 = \left[\begin{array}{ccc|c} & R & & T \\ & & & \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix}. \tag{2.9}$$

The rotational part can be separated into three types of rotations,defined from rotation around the local coordinate frame: $R = R_x(\alpha) R_y(\beta) R_z(\gamma)$, which can be described as these three matrices:

$$R_x(f) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(f) & -sin(f) \\ 0 & sin(f) & cos(f) \end{bmatrix} \tag{2.10}$$

$$R_y(f) = \begin{bmatrix} cos(f) & 0 & -sin(f) \\ 0 & 1 & 0 \\ -sin(f) & 0 & cos(f) \end{bmatrix} \tag{2.11}$$

$$R_z(f) = \begin{bmatrix} cos(f) & -sin(f) & 0 & \\ sin(f) & cos(f) & 0 & \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{2.12}$$

The strength with this approach is that the inverse F^{-1} has the very same format, simply that R is replaced by R^T and T is replaced by $-R^T T$.

2.6 Manipulator kinematics

For manipulators of many kinds it is desirable to know what position and rotation the tip of the manipulator (often referred to as the TCP) has for a given set of joint values. This is known as *forward kinematics*, doing the opposite is simply called, *inverse kinematics*. A generic approach to describe how each joint and arm angle affects the TCP is to use Denavit-Hartenberg parameters.

2.6.1 Denavit-Hartenberg parameters

A Denavit-Hartenberg(DH) matrix is a matrix describing the translation and rotation between two dependent frames. It is a comprehensive approach to model the kinematics in the manipulator, where each axis movement affects the position of all descendant axes.
The DH matrix is composed by four matrices that describes the relationship between frame i and frame $i-1$. These matrices are: a forward rotation matrix R_x, with a rotation α_{i-1} around the x-axis of the previous frame i, a forward translation matrix D_x with the transition a_{i-1} along the previous frame $i-1$'s x-axis, a joint rotation matrix R_z with a rotation θ_i around the frame i's z-axis and a joint translation matrix D_z. Each of these values are defined by the manipulator design, hence are static, except for the joint rotation angle θ_i. This is presented by [12] as follows:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha_{i-1}) & -sin(\alpha_{i-1}) & 0 \\ 0 & sin(\alpha_{i-1}) & cos(\alpha_{i-1}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.13}$$

$$D_x = \begin{bmatrix} 1 & 0 & 0 & a_{i-1} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.14}$$

$$R_z = \begin{bmatrix} cos(\theta_i) & -sin(\theta_i) & 0 & 0 \\ sin(\theta_i) & cos(\theta_i) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.15}$$

$$D_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.16}$$

The compositional result for the transition from frame $i-1$ to the next i is then created :

$$^{i-1}_{\ \ i}DH = R_x(\alpha_{i-1})D_x(a_{i-1})R_z(\theta_i)D_z(d_i), \tag{2.17}$$

where $^{i-1}_{\ \ i}DH$ is the resulting transformation matrix with the format

$$^{i-1}_{\ \ i}DH = \begin{bmatrix} cos\theta_i & -sin\theta_i & 0 & a_{i-1} \\ sin\theta_i cos\alpha_{i-1} & cos\theta_i cos\alpha_{i-1} & -sin\alpha_{i-1} & -d_i sin\alpha_{i-1} \\ sin\theta_i sin\alpha_{i-1} & cos\theta_i sin\alpha_{i-1} & cos\alpha_{i-1} & d_i cos\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.18}$$

Note that the DH notation sometimes differ, sometimes the DH matrix is composed by a rotation and transformation about z and then about x, nevertheless this thesis will use the order present in Equation 2.17. This notation is very much how a frame is defined in Section subsection 2.5.3, which means that the DH matrix also has the same benefit of easy inversion.

2.6.2 Forward kinematics

For a given set of axis values (a pose) the resulting TCP position of the manipulator is to be found. The Denavit-Hartenberg matrix simplifies this to a chain of matrix multiplications, starting with the outer-most axis and a position of origin, the end result being the TCP position denoted p_{TCP}.
The total transformation is summoned by

$$_{f_{TCP}}^{\ \ \ 0}DH = {}^0_1DH\,{}^1_2DH ... {}^{f_{TCP-1}}_{f_{TCP}}DH, \tag{2.19}$$

where the f_{TCP} is the frame located at the TCP. The resulting transformation for the current pose p_{origin} is then

$$p_{TCP} = {}_{f_{TCP}}^{\ \ \ 0}DH p_{origin}, \tag{2.20}$$

where $p_{origin} = [x_0, y_0, z_0, 1]^T$, where the 1 ensures multiplication compatibility with the DH-matrix in Equation 2.18.

2.6.3 Inverse kinematics

The art of achieving angular positions that positions the robots TCP or end effector at a specific point in cartesian space. There is no common generic solution to the inverse and there can be zero to infinitely many solutions depending on manipulator type [6]. In the case of the 6-DOF decoupled manipulator introduced in section 2.1 it is possible to *decouple* the links into positional and orientational [12]. In the case of IRB 6700-175/3.05 with a spherical wrist joint, axes 1, 2 and 3 determine the position and axes 4, 5 and 6 intersects in the point p_{WCP}(located at the WCP), affecting the orientation to the TCP. First, let's define how to get from the base frame to the WCP. The specific inverse kinematic problem in this thesis is described in section 2.7

2.7 Kinematics of IRB 6700-175/3.05

A similar robot, ABB IRB 6620 has been modeled by [13] with a total of nine matrices describing the transformation from the base to the end effector. Since the anatomy of the ABB IRB 6620 and the IRB 6700-175/3.05 share the same properties, just with different dimensions, the approach has much in common. The original concept as such is presented originally by J.Craig in [6], where most of the calculations are derived from.

2.7.1 Denavit-Hartenberg parameters

In Figure 2.3, each frame is represented and the related DH offsets as described in subsection 2.6.1. There are two static offset matrices, independent of axes angles and six matrices that are directly dependent on the manipulator axes. The prettiness with the two static matrices are the possibility to add them separately without involvement in the inverse kinematics. The two static matrices are:

$$ {}^{b}_{0}DH = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_b \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2.21} $$

a offset from the origin to axis 2, and

$$ {}^{6}_{7}DH = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_7 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2.22} $$

the offset from the WCP to the TCP where d_7 is the distance between the WCP and TCP. These two matrices can typically be designed arbitrarily as long as they are static and independent of the joint angles of the manipulator. Additionally there are six matrices that are directly dependent on joint angles:

$$ {}^{0}_{1}DH = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1) & 0 & 0 \\ \sin(\theta_1) & \cos(\theta_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.23} $$

$$ {}^{1}_{2}DH = \begin{bmatrix} \cos(\theta_2) & -\sin(\theta_2) & 0 & a_1 \\ 0 & 0 & 1 & 0 \\ -\sin(\theta_2) & -\cos(\theta_2) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.24} $$

$$ {}^{2}_{3}DH = \begin{bmatrix} \cos(\theta_3) & -\sin(\theta_3) & 0 & a_2 \\ \sin(\theta_3) & \cos(\theta_3) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.25} $$

$$ {}^{3}_{4}DH = \begin{bmatrix} \cos(\theta_4) & -\sin(\theta_4) & 0 & a_3 \\ 0 & 0 & 1 & d_4 \\ -\sin(\theta_4) & -\cos(\theta_4) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.26} $$

$$ {}^{4}_{5}DH = \begin{bmatrix} \cos(\theta_5) & -\sin(\theta_5) & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \sin(\theta_5) & \cos(\theta_5) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.27} $$

$$ {}^{5}_{6}DH = \begin{bmatrix} \cos(\theta_6) & -\sin(\theta_6) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin(\theta_6) & -\cos(\theta_6) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2.28} $$

where the static parameters are presented in the separate Table 2.1.

frame i	$a_i[m]$	$\alpha_i[rad]$	$d_i[m]$
b	0	0	0.35
0	0	0	0
1	0.32	0	0
2	1.14	$-\pi/2$	0
3	0.20	0	0
4	0	$-\pi/2$	1.59
5	0	$\pi/2$	0
6(WCP)	0	$-\pi/2$	0
7(TCP)	0	0	0.21

Table 2.1: DH parameters for the IRB 6700-175/3.05, where frame 7 is the TCP and frame b is the controller world frame, note that values are rounded to two decimals.

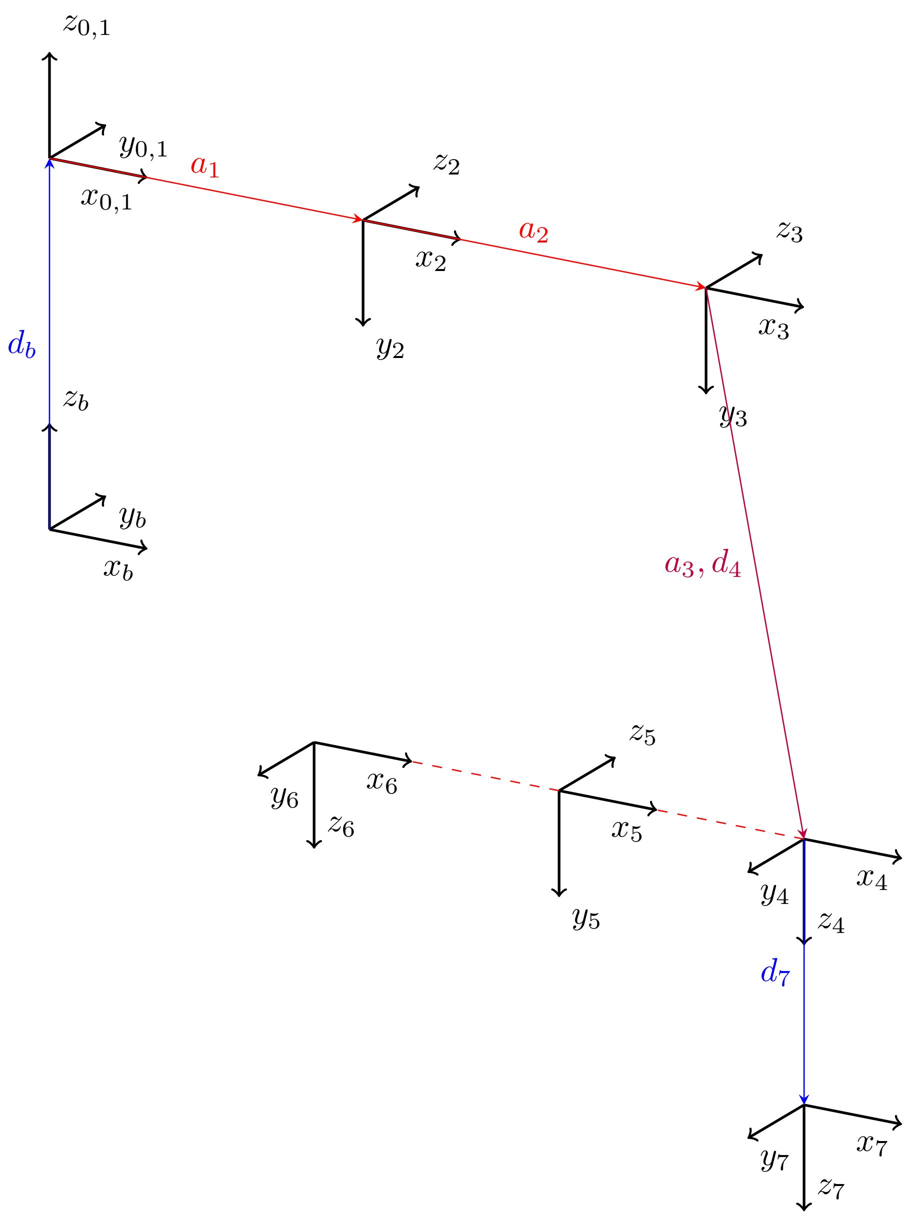

Figure 2.3: All frames of the IRB 6700-175/3.05, where frame b is the base frame and frame 7 the TCP. Each frame i, is represented by a coordinate system defined by x_i, y_i and z_i. The Denavit-Hartenberg translational distances a_i and d_i between each frame i, represents each arm of the manipulator. Note that frame 4,5 and 6 are all positioned in the origin of frame 4 and that frame 0 and 1 shares the same position as well. Dimensions are not proportional.

2.7.2 Inverse kinematics

As described in subsection 2.6.3, the trick to decouple position and orientation will be used to attack the inverse kinematic problem. [6] proposes variable substitution by inverse matrix multiplication to find relationships between each DH matrix and the final position p. The relationship between the end position j and an arbitrary frame i can then be described as

$$ {}^{0}_{i}DH^{-1}\,{}^{0}_{j}DH = {}^{i}_{j}DH, \tag{2.29}$$

which is crucial for this inverse kinematic method.
This section has partly been solved using the programming language python and the symbolic library SymPy [14], the source code developed can be found in section A.3.

Axis 1

$$ {}^{0}_{1}DH^{-1}\,{}^{0}_{6}DH = {}^{1}_{6}DH, \tag{2.30}$$

is expanded to

$$\begin{bmatrix} cos(\theta_1) & sin(\theta_1) & 0 & 0 \\ -sin(\theta_1) & cos(\theta_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} dh_{11} & dh_{12} & dh_{13} & p_x \\ dh_{21} & dh_{22} & dh_{23} & p_y \\ dh_{31} & dh_{32} & dh_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = {}^{1}_{6}DH, \tag{2.31}$$

where dh_{ab} are the expression at column a, row b of the matrix ${}^{0}_{6}DH$. The last column in the right hand side of Equation 2.31 is

$$ {}^{1}_{6}DH_{*,4} = \begin{bmatrix} a_1 + a_2\cos(\theta_2) + a_3\cos(\theta_2+\theta_3) - d_4\sin(\theta_2+\theta_3) \\ 0 \\ -a_2\sin(\theta_2) - a_3\sin(\theta_2+\theta_3) - d_4\cos(\theta_2+\theta_3) \\ 1 \end{bmatrix}. \tag{2.32}$$

Equating cell $(2,4)$, on the lhs and rhs in Equation 2.31 as proposed by [6], boils down to the relationship:

$$ -s_1p_x + c_1p_y = 0, \tag{2.33}$$

which may be simplified to

$$ \theta_1 = \operatorname{atan2}(p_y, p_x) - \operatorname{atan2}(0, \pm 1) = \operatorname{atan2}(p_y, p_x) \pm \pi, \tag{2.34}$$

where $atan2(y,x)$ is a deluxe version of $atan(\frac{y}{x})$, with the addition that it is defined for $x = 0$(as long as $y \neq 0$) and that the signs of x and y are used to find the right quadrant in the unit circle. Hence $atan2(-1,-1) \neq atan(\frac{-1}{-1})$. Also, a cliffhanger for subsubsection 2.7.2.1, the $\pm$ sign gives two solutions to θ_1.

Axis 2

Craig proposes equating $(1,4)$ and $(2,4)$ in the relationship between the transition from frame 0 to frame 3:

$$ {}^{0}_{3}DH^{-1}\,{}^{0}_{6}DH = {}^{3}_{6}DH, \tag{2.35}$$

which gives

$$\begin{bmatrix} c_1c_{2,3} & s_1c_{2+3} & -s_{2,3} & -a_1c_{2,3} - a_2c_3 \\ -s_{2,3}c_1 & -s_1s_{2,3} & -c_{2,3} & a_1s_{2,3} + a_2s_3 \\ -s_1 & c_1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} dh_{11} & dh_{12} & dh_{13} & p_x \\ dh_{21} & dh_{22} & dh_{23} & p_y \\ dh_{31} & dh_{32} & dh_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = {}^{3}_{6}DH, \tag{2.36}$$

where $c_i = cos(\theta_i)$, likewise $s_i = sin(\theta_i)$, in case of a suffix with the structure i, j, it corresponds to $\theta_i + \theta_j$ inside the trigonometric function. The relationship can then be written as

$$\begin{aligned} a_3 &= -a_1 \cos(\theta_2+\theta_3) - a_2 \cos(\theta_3) + p_x \cos(\theta_1)\cos(\theta_2+\theta_3) \\ &\quad + p_y \sin(\theta_1)\cos(\theta_2+\theta_3) - p_z \sin(\theta_2+\theta_3), \qquad (2.37) \\ d_4 &= a_1 \sin(\theta_2+\theta_3) + a_2 \sin(\theta_3) - p_x \sin(\theta_2+\theta_3)\cos(\theta_1) \\ &\quad - p_y \sin(\theta_1)\sin(\theta_2+\theta_3) - p_z \cos(\theta_2+\theta_3), \qquad (2.38) \end{aligned}$$

the first can be solved for $sin(\theta_2 + \theta_3)$ and the second for $cos(\theta_2 + \theta_3)$, by using the common $c_1 p_x + s_1 + y - a_1$ term:

$$s_{(2,3)} = \frac{-p_z(a_2 c_3 + a_3) + (a_2 s_3) - d_4)(-a_1 + p_x c_1 + p_y s_1)}{p_z^2 + (-a_1 + p_x c_1 + p_y s_1)^2}, \qquad (2.39)$$

$$c_{2,3} = \frac{p_z(a_2 s_{(3)} - d_4) - (a_2 c_3 + a_3)(-a_1 + p_x c_1 + p_y s_{(1)})}{p_z^2 + (-a_1 + p_x c_1 + p_y s_1)^2}, \qquad (2.40)$$

finally the common denominator is used to construct one $tan(\theta_2 + \theta_3)$:

$$\theta_2 + \theta_3 = \operatorname{atan}_2(a, b), where \qquad (2.41)$$

$$\begin{gathered} a = p_z(-a_2\cos(\theta_3) - a_3) + (a_2 \sin(\theta_3) - d_4)(-a_1 + p_x\cos(\theta_1) + p_y \sin(\theta_1)) \\ and \\ b = p_z(a_2\sin(\theta_3) - d_4) - (a_2\cos(\theta_3) + a_3)(-a_1 + p_x\cos(\theta_1) + p_y\sin(\theta_1)) \end{gathered}$$

Seen in Equation 2.41, Axis 2 is direct dependent on both axis 1 and 3, However, the opposite is not true, thus, evaluation of axis 2 is done after axis 3.

Axis 3

Since axis 2 is dependent on axis 3 and that the derivation starts with the similar relationship as axis 1, axis 3 is derived before axis 2. From Equation 2.31, the equalities of $(1, 4)$, $(2, 4)$ and $(3, 4)$ are:

$$-a_1 + p_x\cos(\theta_1) + p_y\sin(\theta_1) = a_2\cos(\theta_2) + a_3\cos(\theta_2+\theta_3) - d_4\sin(\theta_2+\theta_3) \qquad (2.42)$$

$$-p_x\sin(\theta_1) + p_y\cos(\theta_1) = 0 \qquad (2.43)$$

$$p_z = -a_2\sin(\theta_2) - a_3\sin(\theta_2+\theta_3) - d_4\cos(\theta_2+\theta_3). \qquad (2.44)$$

Craig [6] proposes a clever trick to eliminate θ_2: square both sides of all three equalities, subtract the lhs with the rhs, finally, add the three equations together and set equal to zero. The end result is the relationship:

$$a_3 \cos(\theta_3) - d_4 \sin(\theta_3) = K, \tag{2.45}$$

$$K = \frac{1}{2a_2}\left(a_1^2 - 2a_1p_x\cos(\theta_1) - 2a_1p_y\sin(\theta_1) - a_2^2 - a_3^2 - d_4^2 + p_x^2 + p_y^2 + p_z^2\right), \tag{2.46}$$

where θ_3 can be derived similarly to Axis 1:

$$\theta_3 = -\operatorname{atan}_2\left(K, \pm\sqrt{K^2 + a_3^2}\right) + \operatorname{atan}_2(a_3, d_4), \tag{2.47}$$

again, note the $\pm$ sign which gives two solutions.

Axis 4

The inverse orientation problem is somewhat lighter in terms of equations, Craig [6]s procedure is followed:
Using the matrix from Equation 2.36 and equating elements $(1,3)$ and $(3,3)$ respectively gives:

$$-\sin(\theta_2 + \theta_3) = -\sin(\theta_5)\cos(\theta_4) \tag{2.48}$$

$$0 = \sin(\theta_4)\sin(\theta_5), \tag{2.49}$$

from which $sin(\theta_4)$ and $cos(\theta_4)$ can be derived to form:

$$\theta_4 = \operatorname{atan}_2(0, \sin(\theta_2 + \theta_3)). \tag{2.50}$$

Though one has to be careful: there is singularity at $sin(\theta_5) = 0$, this happens at the pose where joint 4 and 6 rotates around the same axis, this alters the DH-matrices of the joints, giving them different behavior than what has been modelled. This is in most applications(that includes this thesis) unwanted [6], ABB Robots avoids a singularity at all costs and stops if a singularity has been hit.

Axis 5

$${}^0_4DH^{-1}\,{}^0_6DH = {}^4_6DH, \tag{2.51}$$

which gives

$${}^0_4DH^{-1}\begin{bmatrix} dh_{11} & dh_{12} & dh_{13} & p_x \\ dh_{21} & dh_{22} & dh_{23} & p_y \\ dh_{31} & dh_{32} & dh_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = {}^4_6DH, \tag{2.52}$$

and as for axis 4, equate elements $(1,3)$ and $(3,3)$ and solve for θ_5 results in:

$$\theta_5 = \operatorname{atan}_2(\sin(\theta_2 + \theta_3)\cos(\theta_4), -\cos(\theta_2 + \theta_3)). \tag{2.53}$$

Axis 6

Finally, by using the DH matrix between frame 5 and 6 gives:

$$
{}^{0}_{5}DH^{-1}\,{}^{0}_{6}DH = {}^{5}_{6}DH, \tag{2.54}
$$

which gives

$$
{}^{0}_{5}DH^{-1} \begin{bmatrix} dh_{11} & dh_{12} & dh_{13} & p_x \\ dh_{21} & dh_{22} & dh_{23} & p_y \\ dh_{31} & dh_{32} & dh_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = {}^{5}_{6}DH, \tag{2.55}
$$

this time, equate elements $(1, 1)$ and $(3, 1)$ and solve for θ_6 results in:

$$
\theta_6 = \operatorname{atan}_2(a, b) \tag{2.56}
$$

$$
a = (\sin(\theta_1)\cos(\theta_4) - \sin(\theta_4)\cos(\theta_1)\cos(\theta_2 + \theta_3) \tag{2.57}
$$

$$
\begin{aligned} b = \sin(\theta_1)\sin(\theta 4)\cos(\theta_5) - \sin(\theta_5)\sin(\theta_2 + \theta_3)\cos(\theta_1) \\ + \cos(\theta_1)\cos(\theta_4)\cos(\theta_5)\cos(\theta_2 + \theta_3)). \end{aligned} \tag{2.58}
$$

2.7.2.1 Processing

This could be the finish line, apologies reader, stopping here is like cooking dinner without doing the dishes. Before this section can be left in peace, all configurations need to be found and the desired point to reach p, must be processed.

As noted in Equation 2.34, the $\pm$ contributes to two alternative configurations: θ_1 and θ_1', the same applies to θ_3 in Equation 2.47. Now, one has four solutions, it does not stop here, since the WCP is a spherical joint, axis 4,5 and 6 can be twisted such that the end position is still the same. This contributes to one additional configuration for each of the original four, giving a total of eight. To twist the spherical joint $\theta_4' = \theta_4 + \pi$, $\theta_5' = -\theta_5$ and $\theta_6' = \theta_6 + \pi$. As described in subsection 2.2.1, not all of these configurations are valid because of physical limitations of the axes.

The inverse kinematics would not have been possible without some simplifications made, now the price of that must be paid: The desired end point p where the TCP is desired to be positioned must be pre-processed before the actual inverse kinematic computation can take place. There is one offset in each end of the manipulator, the offset from the base to the imaginary position of axis 1 and the offset from the TCP to the WCP. Additionally, as noted in Figure 2.3, the robot's default position is through the floor, most robots are not designed like this, neither are the IRB 6700-175/3.05. Axis 2's actual 0 position is offset by $-\frac{\pi}{2}$, hence all the true position, denoted by *, is $\theta_2^* = \theta_2 - \frac{\pi}{2}$.

2.8 Linear programming (LP)

Linear programming (LP) is the art of formulating a problem using linear expressions and finding the optimum solution. Combining linear continuous relationships gives a convex set of solutions, hence the local optimum will be the global optimum [15]. A convex solution set is a solution set where no local optimum exists. There are a couple of different numerical, iterative algorithms to find the global optimum, whereas simplex is one of the most common.

LPs can be presented with different notations, to complicate things, the naming of each form differs from author to author. However, *the standard form* is defined by [16] as

$$\begin{aligned} min\ c^T x \\ s.t.\ Ax = b \\ x \geq 0, \end{aligned}$$

where x is the column vector of variables, A a matrix of constants, b a column vector of constants and c a column vector of constants.

2.8.1 Mixed integer linear programming (MILP)

Linear expressions are sometimes limiting problem definitions heavily, some problems might not be formulated at all, a typical example is boolean(true or false) constraints. By introducing binary and integer values, a greater variety of problems may be formulated, producing a mixed integer linear programming (MILP) problem [17]. A MILP problem is very similar to the standard LP problem in its notation, though the mixture of discrete and linear variables summons a completely different type of solution set that rarely is convex. The non-convex solution set limits the strength of classical LP solving methods, though there are various algorithms for finding a solution or an approximation. One simple approximation is to *relax* the MILP problem, solving it as an ordinary LP, when a solution is found, the closest integer solution is chosen, this does not guarantee optimality though. Modern MILP solvers combines many methods to find the optimal solution, though if the solution space is complex and has local optimums, the solution might not be found in polynomial time, which is often denoted as a NP-hard problem [17].

3
Methodology

The approach to the problem is generic, such that only minor modifications and extensions are needed to fit a broad type of single robot cells with a SafeMove configuration. However, focus has been put on a specific single robot cell with a simplified pick and place operation using one trajectory without waiting times.

The algorithm proposed in this thesis consist of two phases; a *preparatory* and an *optimizing* phase. The former phase utilizes RobotStudio to collect data about the safety configuration, brake distances and the programmed trajectory. The latter is fed with the programmed trajectory, with the objective to reduce maximum TCP velocity. The outcome is a positional trajectory similar to the original but with an altered time profile, together with a defined brake margin for each boundary of the safety zone.

3.1 Preparatory phase

Mainly, this algorithm is a final virtual step when the virtual cell configuration and robot trajectory is considered done. The objective is to collect data about the main-trajectory built by the cell engineer and to explore the brake distances of the safety configuration. Finally, presenting the result to the engineer. This phase has been implemented in RobotStudio as a .NET framework C# *add-in* using the publicly available RobotStudio API.

3.1.1 Velocity and cycle time

The robot trajectory is altered with a number of velocity offsets decided by the user, typically three to five steps, each with an offset of 25% is applicable. Each velocity offset is simulated meanwhile poses, maximum TCP velocity and total time (to be the cycle time) is recorded. Angular velocity and angular acceleration is numerically differentiated using forward differentiation. This approach has been chosen mainly to fit the optimizing phase as much as possible where angular positions are the only input-output data. TCP velocity data is produced by feeding the angular velocity of each pose into the forward kinematics matrices.

3.1.2 Zone Definition

The SafeZone is defined by a number of finite-planes with shared intersection vectors. A finite-plane is a normal 3d-plane limited to its two orthogonal spanning vectors v_a

and v_b(orthogonality: $v_a \bullet v_b = 0$) and their intersection point in the 3d-space. The orthogonality of v_a and v_b gives the finite-plane a rectangular shape. The planes are put togeather such that all normal's positive directions is out from the zone, this eases the classification of 3d points as described in section subsection 3.1.3. To create an 'ordinary' fence with a base on the floor, only planes perpendicular to the $xy - plane$ is allowed, However, this condition is not crucial for the algorithm, but an adjustment to fit the SafeMove configuration.

3.1.3 Collision poses

By discretizing each joint's range by a certain resolution r, a discrete joint space with the same dimension as the number of joints is achieved. By exploring the joint space using forward kinematics and collision detection, all poses in the joint space are evaluated and collision poses stored. Since each joint adds a new dimension to the joint space, the number of poses to verify n, grows exponentially with each new joint

$$n = \frac{j_1}{r}\frac{j_2}{r}...\frac{j_N}{r}, \tag{3.1}$$

up to joint N. The algorithm has been restricted to the first three positional joints to limit number of collisions to verify.

Collision detection

By attaching a sphere representing the tool to the TCP, all valid collision poses are filtered out using the following evaluation process:

1. For each pose, compute the end TCP position by forward kinematics as described in section 2.6.
2. Neglect the pose if the TCP position is outside of the safe zone.
3. Using sphere-plane collision detection as in subsection 2.5.2, if there is a collision, save the pose.

This is very much how SafeMove determines during live-supervision if the robot in the discrete time step is colliding or not as described in section 2.4

3.1.4 Violating trajectories

For each collision pose, two zone violating trajectories are created with the purpose to find the trajectory that gives the longest stopping distance. One trajectory is linear, parallel to the collided safe-zone plane's normal. The other trajectory is joint-based, which only moves axis 1. This since according to [2], axis 1 has the longer stopping distances than joint 2 and 3, also, single joint movements are easy to generate without to consider different robot configurations. Mixing these two types of trajectories, covers both the testcases described in [2], and more complex mutli-axis movements.

Linear trajectories

Linear trajectories are complex, mainly since the robot cannot change *configuration* meanwhile travelling a linear trajectory as discussed in subsection 2.2.1. To create a trajectory that is valid and as long as possible (longer possible acceleration distance), the start and endpoints are iteratively found in a n-step algorithm. First, the collision pose is converted to a TCP point using forward kinematics subsection 2.6.2. Then the process for both the end and start positions is as follows:

1. Find the normal to the plane collided with.
2. If number of steps taken is smaller than n, continue, else end.
3. Take a x-long step in the direction of the normal
4. If all of the below is fulfilled, save the new position, increase x by 50% and jump to step 2.
   (a) Use inverse kinematics (described in subsection 2.7.2), is the point reachable?
   (b) Is the configuration compatible with the collision point?
   (c) if $x < 0$: is the point inside the safe zone?
5. If not fulfilled, decrease x by 50%, jump to step 2.

Rotational trajectories, joint 1

Joint-wise trajectories are more natural, less abstract movements, as long as the joint is within its movement range the position is reachable and compatible. The algorithm to find a start and end pose here is done in a similar fashion to the linear trajectory:

1. move joint 1 a $\frac{\pi}{8}$-long step
2. using forward kinematics, calculate the TCP position and verify if it is outside or inside the zone
   (a) when finding a start pose and if the TCP is outside the zone $x := -x$
   (b) when fining the end pose and if the TCP is inside the zone $x := -x$
3. If number of steps taken is smaller than n, continue, else end.
4. Move joint 1 a x-long step
5. If the new position is inside joint limits and less than π, save the value and increase step length by 50% and jump to step 3.
6. else, decrease step length by 50% and jump to step 3.

Collision supervision

For each individual velocity step, brake times and brake distances are recorded. By using the maximum TCP velocity achieved in subsection 3.1.1 all collision trajectories are specified with that velocity. This guarantees that the TCP velocity is lower or at what has been configured, hence there will be scenarios where the TCP velocity is not reached.

To determine the brake distance and time at a collision point, the modified VC described in subsection 2.3.1 is used. To save the data exactly when at the collision pose, collision supervision is used at the virtual boundary. For each zone boundary, only the value with the worst brake distance in terms of how much the boundary was violated is kept. The total violation distance d is is computed by

$$d = |proj_{e,P}| + r, \tag{3.2}$$

where e is the end TCP point for the stop pose, P is the zone boundary plane. The sphere's radius r is also accounted for since that is what collided with the plane, not the TCP.

3.1.5 Cycle time, brake margins

At the end of phase one, the designer is presented by a figure with cycle times on the x-axis and the total brake-margin on the y-axis, an illustration is presented in Figure 3.1. The cell engineer can then either run the approximation procedure again with different velocity steps or choose a cycle time that is acceptable with the related fence distance and proceed with the velocity optimization to decrease the fence margin additionally.

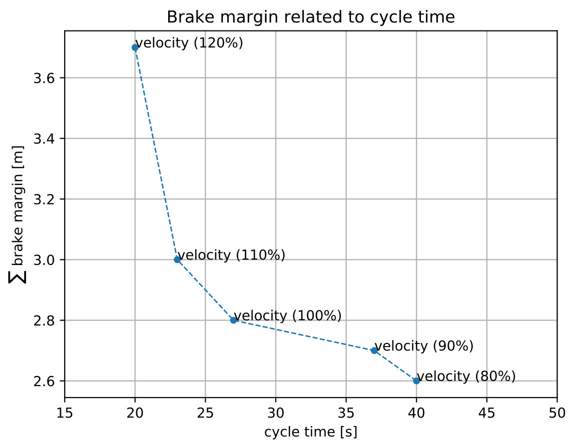

Figure 3.1: An illustration of how the cycle time and brake margin is displayed to the engineer. Where the max brake distance out of each zone boundary has been summed and the total is shown on the y-axis.

3.2 Optimizing phase

The input and output data from the optimizer is a set of poses. To achieve angular velocity and angular acceleration the poses are numerically differentiated using two-point forward differentiation. This adds noise to the data in the first and last value. To eliminate these artifacts: still positions are copied to the beginning and end of the recording, which makes sure that the initial and final velocities are zero.

The optimizing phase has been implemented in `C#` using the linear solving library from Google Operations Research tool [18]. With the vision to one day integrate the two phases of this algorithm into one complete tool. The MILP solver used with the linear solving library is the publicly available open source solver COIN-OR branch and cut [19].

3.2.1 Trajectory optimization

By feeding the trajectory into the optimizer a new position profile can be built by adjusting the time between each target, but not the position order. This optimization is inspired by [4], which has shown good trajectory optimization results using a nonlinear optimization. This approach instead tries to mimic the problem formulation using a discrete up sampling technique to create new data points in-between the originals to create slack. The optimizer now has the freedom to decide when a certain position is reached. The order where the poses are passed through is not changed. Also, stand-still duplicates, where the same pose is present in two adjacent time steps, are identified and the doublette removed. Also joint angles are offset 2π such that all angle positions are strictly positive. When the optimization is complete a decimation process is performed such that the original resolution is preserved.

Constants

Number of unique targets are defined as T, up sampling resolution R, number of axes A and upsampled timesteps N is defined by $N = R(T-1)+1$. Cycle time c_t, is the time difference between the first and the last recorded sample.

Variables and constraints

Each axis is denoted by $a \in [0, A)$. The time indices k ranges $k \in [0, N)$, each pose has an index t, where $t \in [0, T)$. $a, k, t \in \mathbb{Z}$. Each timestep is separated by $\Delta = \frac{c_t}{N}$ Each angular position recorded is stored in

$$\beta_{a,t}. \tag{3.3}$$

The robot pose is defined by its angular position in each joint a and time k

$$\theta_{a,k}, \tag{3.4}$$

angular velocity is directly related to angular position, derived in discrete time:

$$\omega_{a,k} = \frac{\theta_{a,k+1} - \theta_{a,k}}{\Delta}. \tag{3.5}$$

The same relationship is created between angular velocity and angular acceleration:

$$\alpha_{a,k} = \frac{\omega_{a,k+1} - \omega_{a,k}}{\Delta}. \tag{3.6}$$

To connect each pose with a certain time instance the binary variable $p_{t,k}$ is introduced. A second binary variable $q_{t,k}$ with the same dimensions as p is used to indicate a travel between two poses. The relationship between q and p can be described as

$$q_{t,k} = p_{t,k-1} \wedge p_{t,k}, \tag{3.7}$$

which translates to the linear constraints

$$q_{t,k} \geq p_{t,k-1} + p_{t,k} - 1 \forall t, k > 0, \tag{3.8}$$

and

$$q_{t,k} \leq p_{t,k-1} \forall t, k > 0 \tag{3.9}$$

$$q_{t,k} \leq p_{t,k} \forall t, k. \tag{3.10}$$

The travel-between targets variable q, constrains θ to have a value between the previous target $\beta_{a,t}$ and the next target $\beta_{a,t+1}$, described by

$$\theta_{a,k} \leq \beta_{a,t+1}C + \beta_{a,t}(1-C) + M \tag{3.11}$$
$$\theta_{a,k} \geq \beta_{a,t+1}(1-C) + \beta_{a,t}(C) - M \tag{3.12}$$
$$\forall k, a, \ t < T, \tag{3.13}$$

where M is a large penalty constant and $C = max\{sign\{(\beta_{t+1} - \beta_t\}, 0\}$, this is not linear, but allowed since β is constant.

To ensure that only one pose is used at one time instance the following constraint is used

$$\sum_{k}^{N} p_{t,k} = 1 \ \forall t, \tag{3.14}$$

further each pose has to be used at least once, which

$$\sum_{t}^{T} p_{t,k} \geq 1 \ \forall k \tag{3.15}$$

ensures.
Finally, θ is constrained by p and the travel-between targets variable q in a upper bound, for each a, t and k

$$\theta_{a,k} \leq \beta_{a,t} + M(1 - p_{t,k} + q_{t,k}) \forall a, t, k, \tag{3.16}$$

and a lower bound

$$\theta_{a,k} \geq \beta_{a,t} - M(1 - p_{t,k} + q_{t,k}) \forall a, t, k. \tag{3.17}$$

Objectives

Two separate objective functions are proposed, with focus on angular acceleration and velocity. In both objectives presented, the absolute value is desired, to achieve the absolute value, a new variable with only positive values is introduced. Then a lower bound can be put on the variable such that it has to be greater than the original variable and the inverse of the original variable. Since minimization is used, the lack of an upper bound is not a problem.

Formulating TCP velocity using angular velocity and forward kinematics was not possible since forward kinematics includes trigonometric functions, which is not linear. Thus,a other ways much desired TCP velocity objective has been neglected in this thesis.

Minimal summed angular acceleration

Let's use $\gamma_{a,k} \in (0, \infty)$, a linear representation of $|\alpha_{a,k}|$. The objective function can then be formulated as

$$min \sum_{a}^{A} \sum_{k}^{N} \gamma_{a,k}, \tag{3.18}$$

where

$$\gamma_{a,k} \geq \alpha_{a,k} \tag{3.19}$$

and

$$\gamma_{a,k} \geq -\alpha_{a,k}. \tag{3.20}$$

Minimal top angular velocity

Let's introduce $\omega_{max} \in (0, \infty)$, a linear representation of $max(|\omega_{t,k}|)\forall t, k$.

$$min\, \omega_{max}, \tag{3.21}$$

where

$$\omega_{max} \geq \alpha_{a,k} \forall a, k \tag{3.22}$$

and

$$\omega_{max} \geq -\alpha_{a,k} \forall a, k. \tag{3.23}$$

The downside with the angular velocity objective is that it only targets the maximum value, the rest will not be affected. Combining the two objectives with a certain weight to find a good balance between them is an option if the angular velocity objective provides in other terms good results.

3.2.2 Trajectory generation

Since the output from the velocity optimization is joint positions in discrete time these are directly programmed as joint targets, no TCP is specified, instead the time between each target is specified as the original sampling frequency that the poses where collected with initially. This has a consequence; the new trajectory is almost impossible to alter if a design change is wanted. The original trajectory has to be kept.

3.2.3 A final simulation

Finally, a repeated run with the new trajectory to verify that cycle-time is met and to achieve the new top-velocity. This top-velocity is fed a last time to the violating trajectories method and a final run of all trajectories is performed as in subsection 3.1.4. Now the brake distance for each safety boundary is known.

3.2.4 Fence generation

When the brake distance has been found, the final fence position is desired. It is assumed that the fence geometry complexity is identical to the safety boundaries, hence have the same number of vertices. This assumption is restrictive but Using the brake distance for each boundary, a copy of the boundary is generated and offset by the brake distance with the technique described in Equation 2.5.1. When the planes have the right positions, an intersection vertex between each neighboring plane is desired such that the fence geometry can be defined. The intersection vertices are created as described in Equation 2.5.1, where each vertex is positioned on the floor of the cell, thus, the z-value of each vertex is known.

4 Results

The preparatory phase contains several design variables; desired velocity steps to analyze, number of joints to discretize and with what resolution, finally the step length and iterations when generating violation trajectories. Values are provided that has given reasonable results, though a complete test of how much each variable affect the end result has not been evaluated. The same is valid for the only design parameter in the optimizing phase: the up sampling resolution.

4.1 Preparatory phase

A comparison of brake times has been made to verify how the results comply compared with the values provided by ABB in [2]. Also, the relationship between cycle time and the brake distance estimation has been analyzed to see what the relationship looks like for a typical cell.

4.1.1 Brake time and distance

Two test scenarios has been setup to verify and compare the preparatory phases brake distances with those specified by ABB using standard scenarios [2]. Both scenarios were carried out in the same setup, illustrated in Figure 4.1. For comparison, a joint 1-trajectory mimicing that scenario has been created *presented in its raw RAPID form in the appendix*, referred to as *scenario 1.* That scenario is inspired by the one presented by Björn in [3](named scenario 3), since Björn presents similar results when comparing with [2]. The preparatory phase uses a safe zone to find braking distances, *scenario 2* is introduced, where a safe zone is configured such that collision positions will be found in points where the trajectory of *scenario 1* passes through.

The hypothesis is that these scenarios should give similar brake-distances. To ease the comparison, the brake-distance algorithm has been configured to not convert brake angles into brake poses and then into positions but keep the brake angles as is. Both scenarios have been logged in the exact same way.
The result in Figure 4.2 shows an angular brake distance and brake time, greater for **scenario 2** than for **scenario 1**. Hence the proposed algorithm finds trajectories that has longer brake distances for velocities lower than maximum. Note that the behavior for *category 1* brake distances in [2] indicates, that for low velocities, extension zone 0 gives longer brake times and angular brake distances. If the same

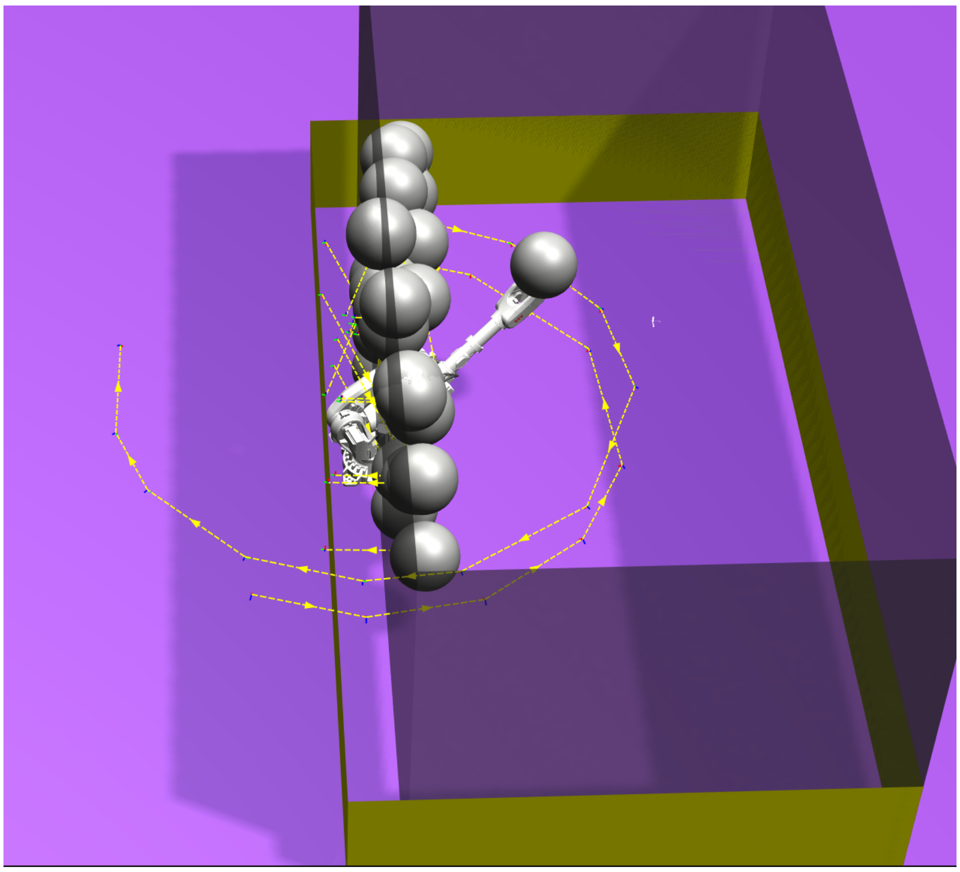

Figure 4.1: The robot cell where two brake distances scenarios where set up. The translucent black border is the safe zone and the yellow border around the cell is the generated fence.

behavior is present at *category 0* stops, that might explain the longer brake distances found by the algorithm. Also note that the numbers in [2] differs greatly from what has been found here. For the IRB 6700-175/3.05, axis 1 *category 0* stop with maximum payload and velocity the brake time is 0.730 *s* compared to 0.280 *s* in Figure 4.2 for *scenario 1*. Comparing the angular brake distance for axis 1 0.714 *rad* with 0.330 *rad* in *scenario 1*, also indicates a great difference in values. These differences might be the result of a perfect simulation world without delays and with optimal brakes and motors.

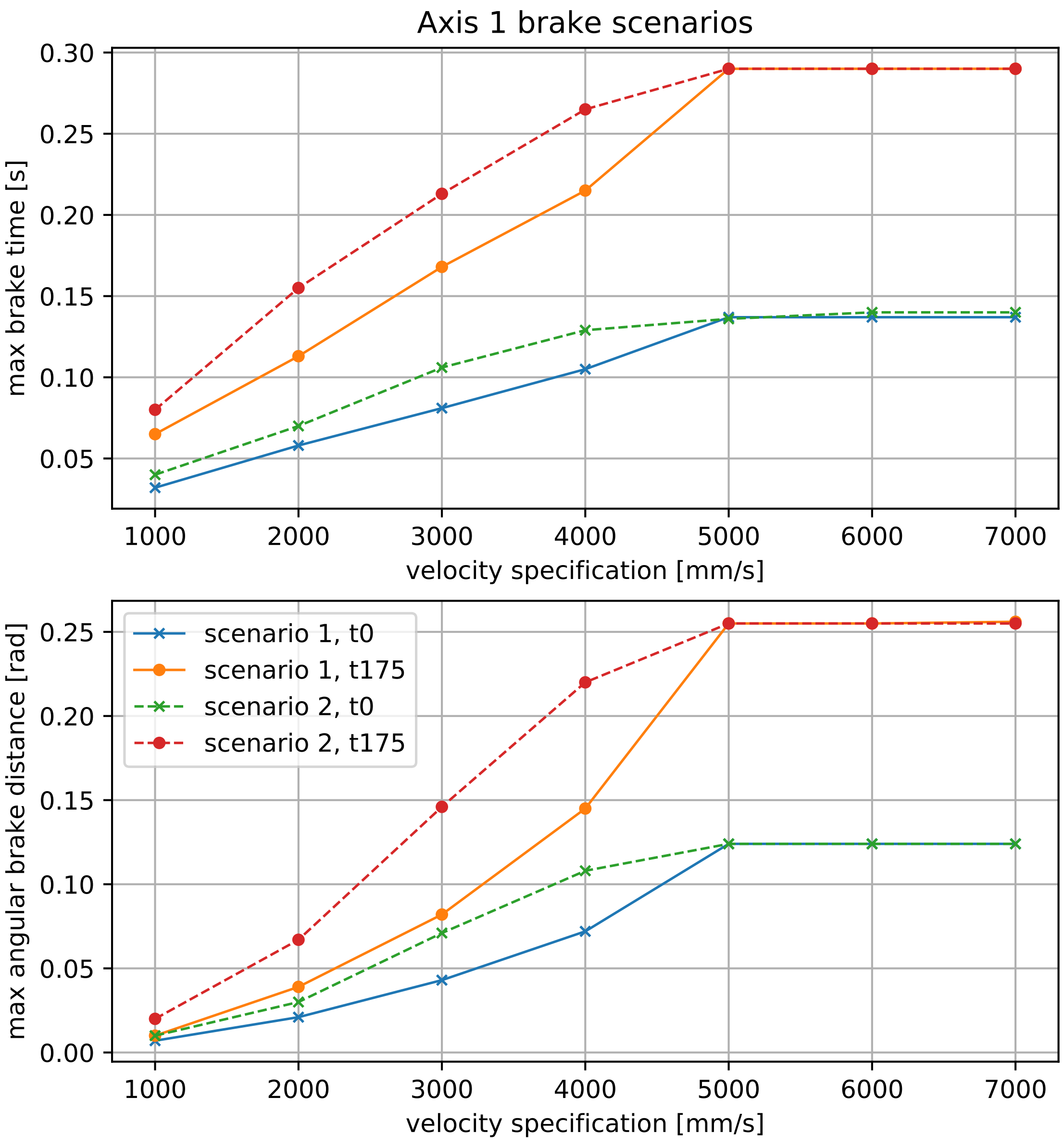

Figure 4.2: Two scenarios with similar setups tested for different velocities and two payloads: 0 kg payload denoted $t0$ and 175 kg payload denoted $t175$.

4.1.2 Cycle time and trajectory velocities

A specific virtual robot cell with a pick and place operation trajectory and an accompanied safe move configuration is used to see what the relationship between cycle time and brake distances are. The cell is shown in Figure 4.3. The relationship graph is shown in Figure 4.4. Even though brake distance out of the virtual safe zone is what is of most importance, it is also interesting to see that brake time very much follows the brake distance.

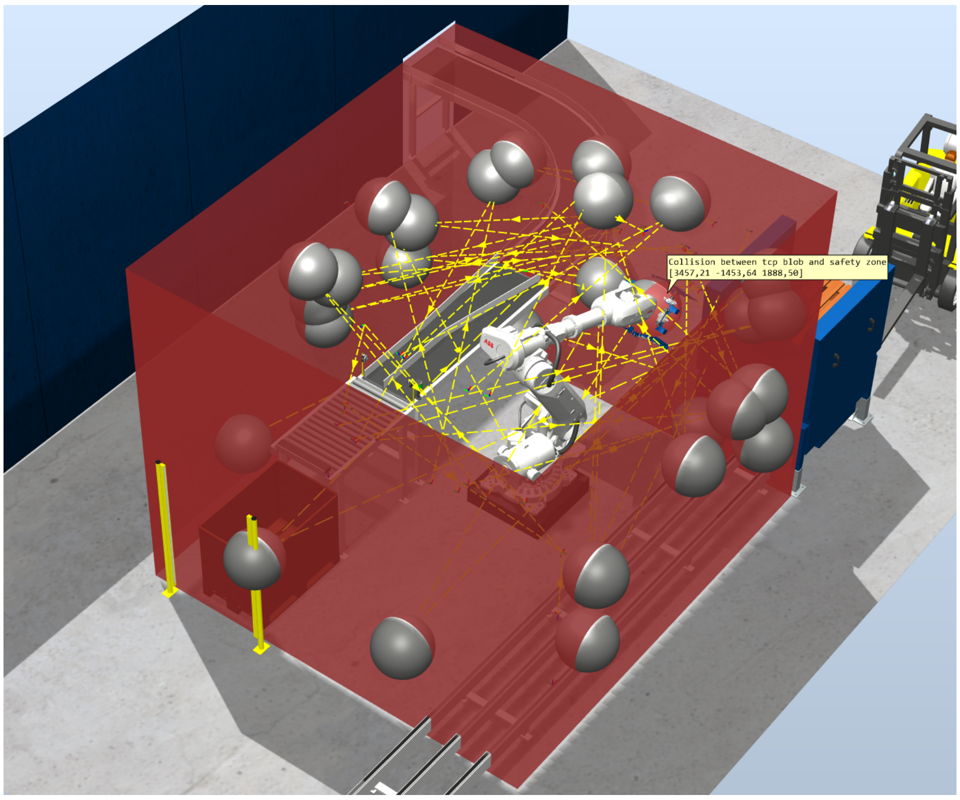

Figure 4.3: A typical pick and place cell with safe move boundaries in red, the yellow dashed lines trajectories and the blobs represent found collision poses

4.2 Optimizing phase

The model proposed in subsection 3.2.1 is used to optimize a recorded robot trajectory using the two different objectives proposed, minimize the summed acceleration and minimize the highest angular velocity. [4] shows a relationship between angular velocity and minimized acceleration. If that relationship can be found here as well it is desired to know if that also lowers TCP velocity.

4.2.1 Minimize acceleration

The same objective was used for two identical trajectories set with different TCP velocities, 600mm/s, shown in Figure 4.5 and 1200mm/s shown in Figure 4.6, the

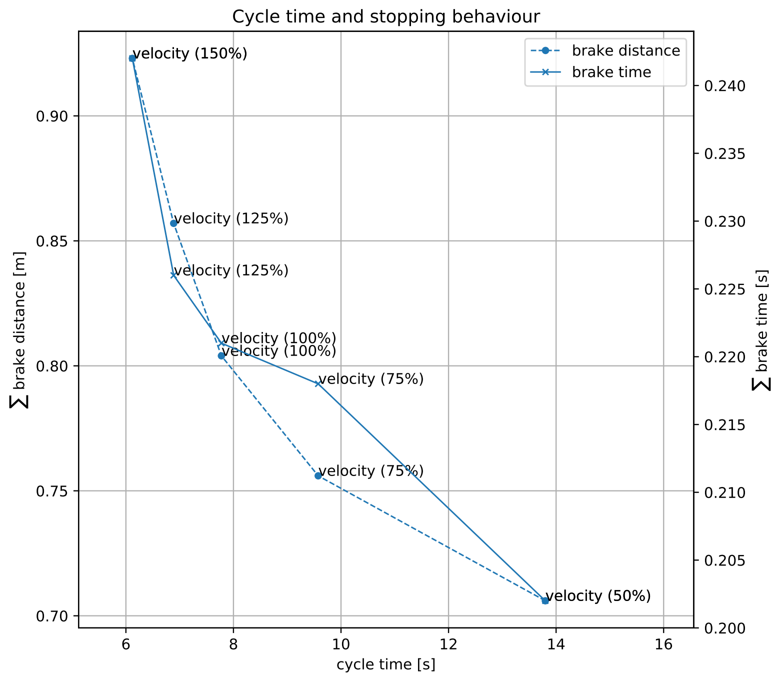

Figure 4.4: A test run with the velocity settings 50%, 75% 100% 125% and 150% for a given trajectory with a velocity of v2000, representing 100% on all targets using the IRB 6700-175/3.05 and the safezone presented in section subsection 4.1.2. Collision poses found using 0.1 rad discretization resolution

total summed acceleration is lower for both trials, so is the top angular velocity However, not the TCP trajectory, even though it has been somewhat altered, top TCP velocity is unchanged.

4.2.2 Minimize max angular velocity

Minimization of max angular velocity as the only objective function did not give any feasible results. Since the objective function only penalizes the very top value of the objective, the solver never converged towards a feasible solution but jumped back and forth between different decision trees. Finally, a result was presented that was fuzzy with huge angular accelerations.

4.2.3 Comparison

Since the whole concept of optimizing trajectories was inspired from the nonlinear model presented in [4], it is of big interest to compare the linear optimization model in this thesis with the original. However, the model provided with this thesis and using the solver Coin-or branch and cut [19] ran for 52 hours with the very same dataset as used in [4], without finding a single feasible solution.

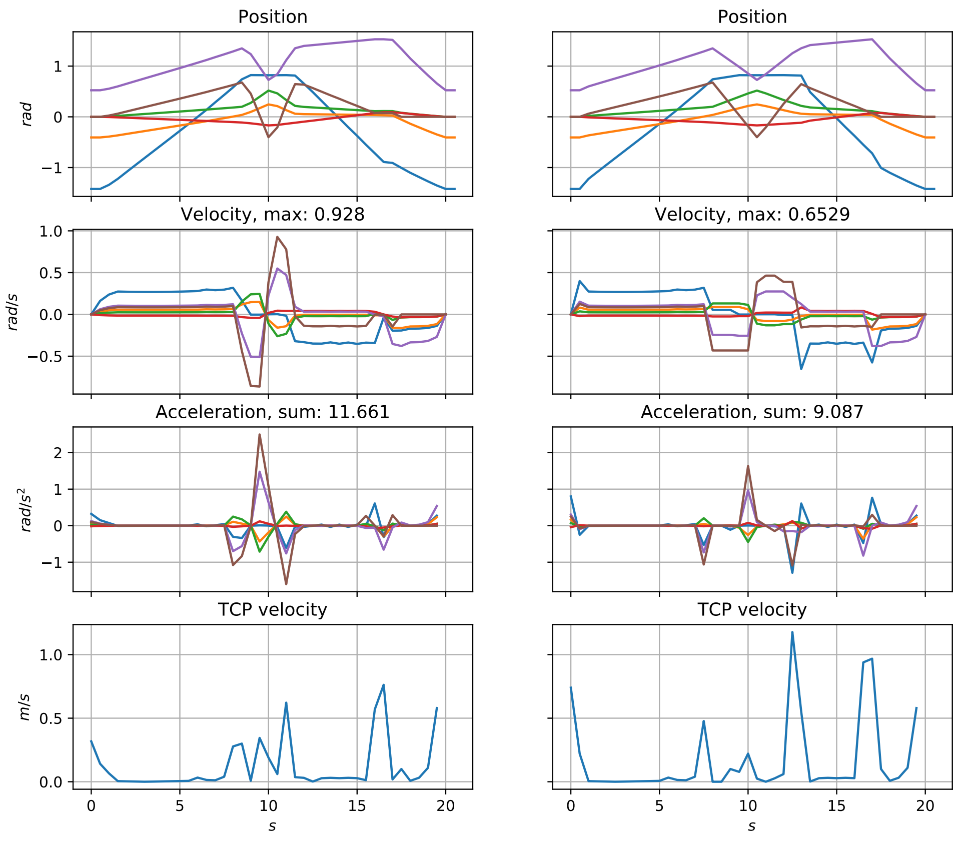

Figure 4.5: Sample time 0.5 s, trajectory velocity set to 600mm/s. The left column is the original recorded trajectory meanwhile the right column is the new optimized trajectory. 3763 variables, 28057 constraints, solved in 22403ms

In general,the performance of the MILP model is far from satisfying, despite that the optimizer is fed with an initial solution, it is very slow at finding a feasible solution. The model description definitely qualifies as a NP-hard problem, although it has been shown for this trajectory with different velocity settings that lower angular velocity does not imply lower TCP velocity.

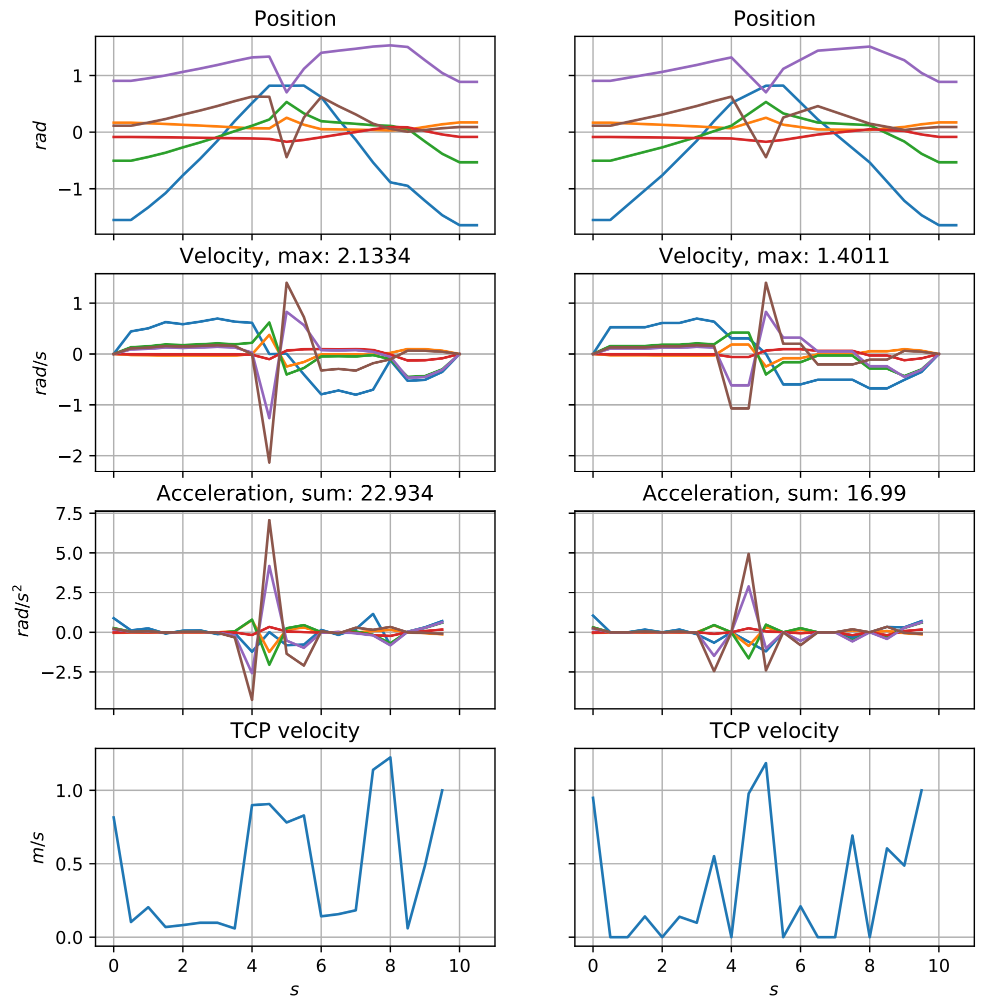

Figure 4.6: Sample time 0.5 *s*, trajectory velocity set to 1200mm/s. The left column is the original recorded trajectory meanwhile the right column is the new optimized trajectory. 2404 variables, 19434 constraints, solved in 17243ms

5 Conclusion

The search for a method to describe cell margins and accompanied cycle-time in a industrial single-robot cell equipped with SafeMove has been successful. The first phase of the algorithm has been integrated into RobotStudio, where it seamlessly collects data about the configuration and trajectories, analyzes and presents cycle times and brake distances for different velocity settings. This algorithm makes it possible for cell engineers to directly analyze and see what tradeoff between cycle time and factory floor area in terms of brake distances that are made. As a bonus, when a desired cycle time is found, one can choose to parse the robot trajectory to the second phase, which minimizes the total acceleration in the trajectory, and the maximum angular velocity, just as [4] has shown. However, for the tested velocities and trajectories max TCP velocity was not lowered and the brake distance could therefore not be decreased.

This thesis has shown that brake distances can with simple algorithms be better approximated than using the standard tabular values. To be more versatile to a greater varieties of robot cells the algorithm needs to be expanded to cover collisions with other geometries on the robot than the TCP such as the upper elbow on the robot close to axis 3. The optimizing phase shows a guidance that simplifying a problem into an integer linear problem neither increase computation time nor improves the result.

Bibliography

[1] ISO ISO. 10218-2: 2011: Robots and robotic devices–safety requirements for industrial robots–part 2: Robot systems and integration. *Geneva, Switzerland: International Organization for Standardization*, 2011.

[2] ABB Robotics. Robot stopping distances according to iso 10218-1. Document ID: 3HAC048645-001, `http://search.abb.com/library/Download.aspx?DocumentID=3HAC048645-001&LanguageCode=en&DocumentPartId=&Action=Launch`, 2017.

[3] Björn Lindqvist. Multi-axis industrial robot braking distance measurements: For risk assessments with virtual safety zones on industrial robots. Master's Thesis at University West, Department of Engineering Science, 2017.

[4] Sarmad Riazi, Kristofer Bengtsson, Rainer Bischoff, Andreas Aurnhammer, Oskar Wigström, and Bengt Lennartson. Energy and peak-power optimization of existing time-optimal robot trajectories. In *Automation Science and Engineering (CASE), 2016 IEEE International Conference on*, pages 321–327. IEEE, 2016.

[5] ABB Robotics. Technical reference manual - rapid instructions, functions and data types. Document ID: 33HAC050917-001 Revision: G, 2018.

[6] John J Craig. *Introduction to robotics: mechanics and control*, volume 3. Pearson/Prentice Hall Upper Saddle River, NJ, USA:, 2005.

[7] Realistic Robot Simulation II. In *VRC-Interface Specification Version 1.0*, pages 1–2. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), VRC-Specification-Maintenance Management, 2001.

[8] ABB Robotics. Applicationmanualfunctionalsafetyand safemove2. Document ID:3HAC052610-001 Revision: G, 2018.

[9] Robert A. Adams and Christopher Essex. *Calculus: a complete course.* Pearson, Toronto, 8. edition, 2013.

[10] Eric W. Weisstein. "plane-plane intersection.". From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/Plane-PlaneIntersection.html`, 2018.

[11] Christer Ericson. *Real-time collision detection.* CRC Press, 2004.

[12] Jorge Angeles. *Fundamentals of robotic mechanical systems*, volume 2. Springer, 2002.

[13] Adrian-Florin Nicolescu, Florentin-Marian Ilie, and Tudor-George Alexandru. Forward and inverse kinematics study of industrial robots taking into account constructive and functional parameter's modeling. *Proceedings in Manufacturing Systems*, 10(4):157, 2015.

[14] O Certik et al. Sympy python library for symbolic mathematics, 2008.

[15] George Dantzig. *Linear programming and extensions.* Princeton university press, 2016.

[16] Howard Karloff. *Linear programming.* Springer Science & Business Media, 2008.

[17] Frédérico Della Croce. Mixed integer linear programming models for combinatorial optimization problems, 2014.

[18] Google Inc. Google operations research tool. `https://developers.google.com/optimization/`, 2018.

[19] John Forrest. Cbc (coin-or branch and cut) open-source mixed integer programming solver. `https://projects.coin-or.org/Cbc`, 2018.

A
Appendix

- Robotstudio stations, RAPID trajectories and recorded trajectories published at: `github.com/ohenriksson/MastersThesisData`
- Symbolic kinematic solver, forward kinematics and brake distance plotting : `github.com/ohenriksson/python-robot-kinematics`

A.1 Scenario based brake test

where *vmax* is replaced with $[vel, 5000, 5000, 5000]$, where the desired velocity *vel* is defined in mm/s.

A.1.1 Scenario 1 movements

Also available at `github.com/ohenriksson/MastersThesisData/scenario1.RAPID`

```
CONST robtarget Target_10:=[[2650.778985035,-1514.394022891,665.53670734],[0.237207259,0.063532185,0.962557826,-0.114798994],[-1,0,-1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
CONST robtarget Target_20:=[[3044.708925288,-223.083043607,665.53670734],[0.256719524,-0.150874772,0.952780515,-0.059507803],[-1,0,-1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
CONST robtarget Target_30:=[[2830.995781706,1142.576635519,665.53670734],[0.263526219,-0.362657835,0.893886571,0.000156031],[0,0,-1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
CONST robtarget Target_40:=[[1941.519398723,2355.954351946,665.53670734],[0.255362535,-0.571733158,0.776965317,0.065085085],[0,0,-1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
CONST robtarget Target_50:=[[758.480316395,2957.148304315,665.53670734],[0.235112529,-0.726933226,0.634130896,0.119030212],[0,0,-1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
CONST robtarget Target_60:=[[-1267.010409106,2777.53543763,665.53670734],[0.182049336,-0.896624462,0.355834908,0.190536432],[1,0,-1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
CONST robtarget Target_70:=[[-2708.612226105,1408.345942114,665.53670734],[0.110083745,-0.963626329,0.044477023,0.239431956],[1,0,-1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
CONST robtarget Target_80:=[[-3006.534548987,529.875919266,665.53670734],[0.072430236,-0.959197766,-0.102438045,0.253377098],[1,0,-1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];

CONST jointtarget JointTarget_5:=[[48.429291781,85,-98.742260129,0.112841515,40.027258121,-22.258237543],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
CONST jointtarget JointTarget_6
    :=[[15.425917994,85,-98.742260129,0.112841515,40.027258121,-22.258237543],[9E
    +09,9E+09,9E+09,9E+09,9E+09,9E+09]];
CONST jointtarget JointTarget_7
    :=[[-5.037431675,85,-98.742260129,0.112841515,40.027258121,-22.258237543],[9E
    +09,9E+09,9E+09,9E+09,9E+09,9E+09]];
CONST jointtarget JointTarget_8
    :=[[-30.266766578,85,-98.742260129,0.112841515,40.027258121,-22.258237543],[9
    E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
CONST jointtarget JointTarget_9
    :=[[-58.818282132,85,-98.742260129,0.112841515,40.027258121,-22.258237543],[9
    E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
CONST jointtarget JointTarget_10
    :=[[-77.492415908,85,-98.742260129,0.112841515,40.027258121,-22.258237543],[9
    E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
CONST jointtarget JointTarget_11
    :=[[-105.198837588,85,-98.742260129,0.112841515,40.027258121,-22.258237543],[9
    E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];

PROC Path_10()
MoveAbsJ JointTarget_1,vmax,z200,tool0\WObj:=wobj0;
MoveAbsJ JointTarget_2,vmax,z200,tool0\WObj:=wobj0;
MoveAbsJ JointTarget_3,vmax,z200,tool0\WObj:=wobj0;
MoveAbsJ JointTarget_4,vmax,z200,tool0\WObj:=wobj0;
MoveAbsJ JointTarget_5,vmax,z200,tool0\WObj:=wobj0;
MoveAbsJ JointTarget_6,vmax,z200,tool0\WObj:=wobj0;
MoveAbsJ JointTarget_7,vmax,z200,tool0\WObj:=wobj0;
MoveAbsJ JointTarget_8,vmax,z200,tool0\WObj:=wobj0;
MoveAbsJ JointTarget_9,vmax,z200,tool0\WObj:=wobj0;
MoveAbsJ JointTarget_10,vmax,z200,tool0\WObj:=wobj0;
MoveAbsJ JointTarget_11,vmax,z200,tool0\WObj:=wobj0;
ENDPROC
```

A.1.2 Scenario 2 movements

Scenario 2 is quite long, instead the whole code is available at `github.com/ohenriksson/MastersThesisData/scenario2.RAPID`

A.2 Trajectory optimization

Trajectory used to optimize upon, also available at `github.com/ohenriksson/MastersThesisData/cycle-time-trajectory.RAPID`

```
CONST robtarget home:=[[295.012360665,-1961.528973741,2549.551272427],
    [0.503498848,0.487616868,0.566438135,-0.433435739],[-1,-1,0,0],[9E+09,9E+09,9
    E+09,9E+09,9E+09,9E+09]];
CONST robtarget pick_appr:=[[1225.117136908,-1433.471650837,1173.632948636],
    [0.028438528,-0.437587093,-0.896246668,-0.06671354],[-1,0,-1,0],[9E+09,9E
    +09,9E+09,9E+09,9E+09,9E+09]];
CONST robtarget pick:=[[1173.225075815,-1471.482028347,1003.1900711],
    [0.021718246,-0.72642022,-0.686503609,-0.023553701],[-1,0,0,0],[9E+09,9E+09,9
    E+09,9E+09,9E+09,9E+09]];
```

```
CONST robtarget place_appr:=[[1367.611678682,1352.581913875,970.575639752],
    [0.013576519,-0.072386312,0.995149194,-0.065222558],[0,-1,0,0],[9E+09,9E+09,9
    E+09,9E+09,9E+09,9E+09]];
CONST robtarget place:=[[1367.61201179,1352.581938252,93.612530794],
    [0.059786579,-0.693245635,0.716568642,-0.048635744],[0,-1,-1,0],[9E+09,9E
    +09,9E+09,9E+09,9E+09,9E+09]];
PROC Path_10()
    MoveJ home,[600, 500, 5000, 1000],z100,tGripper\WObj:=wobj0;
    MoveJ pick_appr,[600, 500, 5000, 1000],z100,tGripper\WObj:=wobj0;
    MoveJ pick,[600, 500, 5000, 1000],z100,tGripper\WObj:=wobj0;
    MoveJ pick_appr,[600, 500, 5000, 1000],z100,tGripper\WObj:=wobj0;
    MoveJ place_appr,[600, 500, 5000, 1000],z100,tGripper\WObj:=wobj0;
    MoveJ place,[600, 500, 5000, 1000],z100,tGripper\WObj:=wobj0;
    MoveJ place_appr,[600, 500, 5000, 1000],z100,tGripper\WObj:=wobj0;
    MoveJ home,[600, 500, 5000, 1000],z100,tGripper\WObj:=wobj0;
ENDPROC
```

A.3 Symbolic kinematic python solver

The symbolic kinematic solver and brake distance plotter is also available at `github.com/ohenriksson/python-robot-kinematics`

```
from sympy import *
from sympy.abc import *
import mpmath
import numpy

mpmath.mp.pretty = True
init_printing(use_unicode=True)

class c(Function):
    @classmethod
    def eval(cls,x):
        # return cos(x)
        if x == pi/2: return 0
        if x == -pi/2: return 0
        if x == 0: return 1
        else:
            return cos(x)

class s(Function):
    @classmethod
    def eval(cls,x):
        # return sin(x)
        if x == pi/2: return 1
        if x == -pi/2: return -1
        if x == 0: return 0
        else:
            return sin(x)

class b(Function): #alfa
    @classmethod
    def eval(cls,x):
        if x.is_Number:
            if x == 1: return -pi/2
            elif x == 3: return -pi/2
            elif x == 4: return pi/2
            elif x == 5: return -pi/2
            else: return 0

class a(Function):
    @classmethod
    def eval(cls,x):
        if x.is_Number:
            if x == 1: return Symbol('a1')
            elif x == 2: return Symbol('a2')
            elif x == 3: return Symbol('a3')
            else: return 0
        if x.is_String:
            return Symbol('a'+x)

class d(Function):
    @classmethod
    def eval(cls,x):
        if x.is_Number:
            if x == 4: return Symbol('d4')
            elif x == 7: return Symbol('d7')
            else: return 0
        else: return Symbol('d_'+str(x))

class t(Function):
    @classmethod
    def eval(cls,x):
        if x.is_Number:
            if x == 0: return 0
            else: return Symbol(str(theta) +str(x))
        else: return 0

def getmatrix(i, dval=0):
    if dval == 0: dval = d(i)
    else: dval = d(dval)
    mat = Matrix([
        [c(t(i)), -s(t(i)), 0, a(i-1)],
        [s(t(i))*c(b(i-1)), c(t(i))*c(b(i-1)), -s(b(i-1)), -dval*s(b(i-1))],
        [s(t(i))*s(b(i-1)), c(t(i))*s(b(i-1)), c(b(i-1)), dval*c(b(i-1))],
        [0,0,0,1]
    ])
    return mat

def MyInv(matrix):
    myinv = matrix
    A = matrix[0:3,0:3]
    R = matrix[0:3,3]
    Rnew = -(A.T)*R
    myinv[0:3,0:3] = simplify(A.T)
    myinv[0:3,3] = Rnew
    return myinv
```

```python
def printall(list1,indices):
    for x,i in zip(list1,indices):
        str1 = '\\begin{equation}␣\prescript{' + i[0] +'}{' +i[1] +'}{DH}␣=␣'
        str2 = '\end{equation}'
        print(str1 + latex(x) +str2)
        pprint(x)

def createPosMatrix():
    x, y, z = symbols('p_x␣p_y␣p_z')
    Pmat = eye(4)
    Pmat[0:3,3] = Matrix([[x,y,z]]).T
    return Pmat

def solveTheta1(T01, T16):
    print('-----theta1-------')
    leftM = MyInv(T01)*createPosMatrix()
    py = T16.row(1).col(3)[0]
    temp1 = Eq(leftM.row(1).col(3)[0],py)
    theta1 = solve(temp1,t(1))
    print(latex(theta1[0]))
    print(latex(theta1[1]))

def solveTheta3(T01, T16):
    p_x, p_y, p_z = symbols('p_x␣p_y␣p_z')
    print('-----theta3-------')
    leftM = MyInv(T01)*createPosMatrix()
    px = T16.row(0).col(3)[0]
    py = T16.row(1).col(3)[0]
    pz = T16.row(2).col(3)[0]
    l14 = leftM.row(0).col(3)[0]
    l24 = leftM.row(1).col(3)[0]
    l34 = leftM.row(2).col(3)[0]

    #the solver could not solve it with a on the wrong side of the equal sign(
        probably because of the squaring later)
    l14 = l14 -a(1)
    px = px - a(1)

    eqx = simplify(Eq(l14,px)**2)
    eqy = simplify(Eq(l24,py)**2)
    eqz = simplify(Eq(l34,pz)**2)

    #sadly, squaring the whole Eq did not follow through, workaround:
    eqx = pow(l14,2)-pow(px,2)
    eqy = pow(l24,2)-pow(py,2)
    eqz = pow(l34,2)-pow(pz,2)

    eq4 = Eq(simplify(eqx+eqy+eqz),0)
    pprint(eq4)
    lhs = 2*a(2) * (a(3)*c(t(3)) - d(4)*s(t(3)))
    rhs = a(1)**2 - a(3)**2 -d(4)**2 + p_x**2 + p_y**2 -a(2)**2 + p_z**2 - 2*a(1)
        *p_x*c(t(1)) - 2*a(1) *p_y *s(t(1))
    lhs = solve(eq4,rhs)[0]

    lhs = lhs/(2*a(2))
    K = rhs/(2*a(2))
    # slask = solve(eq4, slask = solve(eq4, (d(4)*s(t(3))) )
    print(latex(lhs))
    print(latex(K))
    pprint(lhs)
    pprint(K)
    K = symbols('K')

    theta3a = atan2(a(3),d(4)) - atan2(K,+sqrt(a(3)**2+d(3)**2-K**2))
    theta3b = atan2(a(3),d(4)) - atan2(K,-sqrt(a(3)**2+d(3)**2-K**2))
    print(latex(theta3a))
    return theta3a


def solveTheta2(T03, T36):
    p_x, p_y, p_z = symbols('p_x␣p_y␣p_z')
    print('-----theta2-------')
    leftM = MyInv(T03)*createPosMatrix()
    l14 = leftM.row(0).col(3)[0]
    r14 = T36.row(0).col(3)[0]
    l24 = leftM.row(1).col(3)[0]
    r24 = T36.row(1).col(3)[0]
    eq1 = l14-r14
    eq2 = l24-r24

    denumA = c(t(1))*p_x + s(t(1))*p_y -a(1)
    denum = p_z**2 + denumA**2

    nom1 = p_z*(-a(3)-a(2)*c(t(3))) + denumA*(a(2)* s(t(3))-d(4))
    eq1 = Eq(s(t(2)+t(3)),nom1/denum)
    nom2 = p_z*(a(2)*s(t(3))-d(4)) - denumA*(a(3) + a(2)*c(t(3)))
    eq2 = Eq(c(t(2)+t(3)),nom2/denum)

    eq1 = (simplify(eq1))
    eq2 = (simplify(eq2))
    print(latex(eq1))
    print(latex(eq2))
    print('---')

    EQ = Eq(t(2)+t(3),atan2(nom1,nom2))
    print(latex(EQ))
    pprint(EQ)

    return
    # c1px = solve(eq1,c(t(1)*p_x)))[0]
    # c23 = solve(eq2,sin(t(2)+t(3)))[0]
    # pprint(simplify(s23))
    # pprint(simplify(c23))
    # print('---')

    # eq2b = eq2.subs(sin(t(2)+t(3)),s23)
    # pprint(simplify(eq2b))
    # return

    # A, B = fraction(solve(eq1,c(t(2)+t(3)))[0])
    # C, B = fraction(solve(eq2,s(t(2)+t(3)))[0])
    # #assume z>0
    # theta23 = atan2(A,C)
    # theta2 = theta23 - t(3)
    # pprint(theta23)
    # print('----')
    # pprint(latex(Eq(t(2)+t(3),theta23)))
    # print(latex(Eq(t(2)+t(3),theta23)))
    # return theta2


def solveTheta4(T03, T36):
    print('-----theta4-------')
    leftM = MyInv(T03)*createPosMatrix()
    c = 2
    l13 = leftM.row(0).col(c)[0]
    r13 = T36.row(0).col(c)[0]
    l33 = leftM.row(2).col(c)[0]
    r33 = T36.row(2).col(c)[0]
    eq1 = Eq(l13,r13)
    eq2 = Eq(l33,r33)

    print(latex(eq1))
    print(latex(eq2))

    c4 = solve(eq1, cos(t(4))*sin(t(5)))[0]
    s4 = solve(eq2, sin(t(4))*sin(t(5)))[0]

    #assume s5!=0 #singularity
    theta4 = Eq(t(4),atan2(s4,c4))
    pprint(theta4)
    print(latex(theta4))
    return theta4


def solveTheta5(T04, T46):
    print('-----theta5-------')
    leftM = MyInv(T04)*createPosMatrix()
    c = 2
    l13 = leftM.row(0).col(c)[0]
    r13 = T46.row(0).col(c)[0]
    l33 = leftM.row(2).col(c)[0]
    r33 = T46.row(2).col(c)[0]
    s5 = solve(l13-r13,sin(t(5)))[0]
    c5 = solve(l33-r33,cos(t(5)))[0]
    theta5 = Eq(t(5),atan2(s5,c5))

    pprint(theta5)
    print(latex(theta5))

    return theta5


def solveTheta6(T05, T56):
    print('-----theta6-------')
    leftM = MyInv(T05)*createPosMatrix()
    c = 0
    l11 = leftM.row(0).col(c)[0]
    r11 = T56.row(0).col(c)[0]
    l31 = leftM.row(2).col(c)[0]
    r31 = T56.row(2).col(c)[0]
    c6 = solve(l11-r11,cos(t(6)))[0]
    s6 = solve(l31-r31,sin(t(6)))[0]
    theta6 = Eq(t(6),atan2(s6,c6))

    pprint(theta6)
    print(latex(theta6))
    return theta6
```