Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Thea Holmedal

Implementing robotic offline programming with the Yaskawa Motoman GP25-12

Master's thesis in Mechanical Engineering
Supervisor: Lars Tingelstad
June 2021

NTNU
Norwegian University of
Science and Technology

Thea Holmedal

Implementing robotic offline programming with the Yaskawa Motoman GP25-12

Master's thesis in Mechanical Engineering
Supervisor: Lars Tingelstad
June 2021

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

NTNU
Norwegian University of
Science and Technology

Preface

This Master´s thesis represents the concluding work of a five-year integrated master degree at The Department of Mechanical and Industrial Engineering, with a specialization within Robotics and Automation. This thesis has been developed during spring 2021, while it reflects this final year in total, incorporating insight obtained during the fall of 2020 as well.

The robotic laboratory at Perleporten, NTNU, has been a vital part of this project. I want to thank NTNU for providing such excellent facilities, allowing their students to experiment and test the theory in practice.

I want to express my gratitude to my supervisor Lars Tingelstad for allowing me to explore robotic offline programming with his guidance, knowledge, and patience. I would also like to thank Vebjørn Bergsholm Bjørhovde for the many valuable discussions we had throughout the semester.

Lastly, I want to thank my fellow students on the 5th floor at Perleporten. With COVID-19 and the circumstances being what they are, the positive and light-spirited environment has been invaluable.

Summary

The manufacturing industry is changing. Products are being produced faster, demands are changing faster, and personalized items are desired. This is enabled by, e.g., cyber-physical systems, the Internet of Things, machine learning, and artificial intelligence, and represents such a significant era that it is being referred to as the fourth industrial revolution, or, Industry 4.0

Industry 4.0 differs from Industry 3.0 with how systems are digitally connected and can share and receive information constantly. The ultimate goal is autonomous systems capable of making decisions based on the received information.

A major challenge lies within making the existing infrastructure and systems interoperable, where the open platform communication OPC UA has been suggested as the solution. While the OPC UA protocol may solve the problem of interoperability, the automation industry is faced with a different problem; the rapid change in demand.

Robotic automation was a vital part of Industry 3.0 as it enabled mass-production, providing an efficiency not possible to achieve with human labour. However, the new demands introduced with Industry 4.0 requires a change in the robotic automation industry; it must adapt to a high-mix, low-volume production.

Robots are mainly programmed in two ways; online or offline. Online programming directly involves the robot while programming and therefore requires production to stop. Robotic offline programming is a method for programming robots from a computer, allowing for production to continue. The reduction in production downtime is an invaluable advantage with offline programming, which essentially enables production compatible with the requirements posed by Industry 4.0.

This thesis concerns the implementation of offline programming with the robot manipulator Yaskawa Motoman GP25-12. The simulation software Visual Components was used to design a digital representation of the robot cell at Perleporten and program a trajectory to be sent to the physical robot. An OPC UA server was developed in order to communicate with the simulation software. The third and final software implemented was the Moto library, used to communicate with

the robot controller.

Two methods of implementation were developed. The first method incorporates all three software components in one Python script that allows for sending trajectory points directly to the robot controller from Visual Components while the simulation is running. The second method writes trajectory points from simulation to file, where a separate Python script was developed for sending the trajectory points to the robot controller. The success of the first method proved difficult to analyze as it, with the suggested solution, was not possible to obtain joint feedback from the physical robot.

The success of the second method was examined by comparing the trajectory from simulation with the trajectory performed by the physical robot. Different configurations within the scripts were tested. The best result displayed minor deviations between the offline programmed trajectory and the trajectory performed by the physical robot.

The thesis has shown that the implementation of offline programming with the Yaskawa Motoman GP25-12 is possible. However, the suggested solutions need modifications. Further work includes solving the issue of speed in the script encountered when sending trajectory points to the robot controller and the requirement for the physical robot to be in its zero position before receiving trajectory points.

Sammendrag

Produksjonsindustrien er i endring. Produkter produseres raskere, etterspørsel endres raskere og tilpassede varer er ettertraktet. Dette er muliggjort av for eksempel cyber-fysiske systemer, tingenes internett, maskinlæring og kunstig intelligens, og representerer en så betydelig epoke at den blir referert til som den fjerde industrielle revolusjonen, eller Industri 4.0

Industri 4.0 skiller seg fra Industri 3.0 med hvordan systemer er koblet digitalt og kontinuerlig kan dele og motta informasjon. Det endelige målet er autonome systemer som er i stand til å ta beslutninger basert på mottatt informasjon.

En stor utfordring ligger i det å gjøre eksisterende infrastruktur og systemer interoperable, der kommunikasjonsprotokollen OPC UA er blitt foreslått som løsningen. Mens OPC UA protokollen kan løse problemet med interoperabilitet, står automatiseringsindustrien overfor et annet problem; den raske endringen i etterspørsel.

Robotautomatisering var en viktig del av Industri 3.0, da den muliggjorde masseproduksjon, og ga en effektivitet som ikke var mulig å oppnå med menneskelig arbeidskraft. Imidlertid krever endringen i etterspørsel, introdusert med Industri 4.0, at robotindustrien tilpasser seg; den må kunne implementeres i produksjon med stor ending og lavt volum.

Roboter er hovedsakelig programmert på to måter; online eller offline. Online programmering involverer roboten direkte mens den programmeres, og krever derfor at produksjonen stopper. Offline programmering er en metode for programmering av roboter fra en datamaskin, slik at produksjonen kan fortsette. Reduksjonen i nedetid for produksjonen er en uvurderlig fordel med offline programmering, som i hovedsak muliggjør produksjon som er kompatibel med kravene fra Industry 4.0.

Denne oppgaven gjelder implementering av offline programmering med robot manipulatoren Yaskawa Motoman GP25-12. Simuleringsverktøyet Visual Components ble brukt til å designe en digital representasjon av robotcellen på Perleporten og for å programmere en bane som skulle implementeres med den fysiske roboten. En OPC UA-server ble utviklet for å kommunisere med simuleringsverktøyet. Den tredje og siste programvaren som ble implementert var Moto-biblioteket, brukt til

å kommunisere med robotkontrolleren.

To implementeringsmetoder ble utviklet. Den første metoden inneholder alle de tre komponentene i ett Python-skript som gjør det mulig å sende banepunkter direkte til robotkontrolleren fra Visual Components mens simuleringen kjører. Den andre metoden skriver banepunkter fra simulering til fil, hvor det ble utviklet et eget Python-skript for å sende banepunktene til robotkontrolleren. Hvorvidt den første metoden var suksessfull eller ikke, viste seg vanskelig å analysere da det med den foreslåtte løsningen, ikke var mulig å lese leddposisjonene til den fysiske roboten mens den beveget seg.

Suksessen til den andre metoden ble undersøkt ved å sammenligne banen fra simulering med banen som ble utført av den fysiske roboten. Ulike konfigurasjoner i skriptene ble testet. Det beste resultatet viste mindre avvik mellom den offline-programmerte banen og banen som ble utført av den fysiske roboten.

Oppgaven har vist at implementering av offline programmering med Yaskawa Motoman GP25-12 er mulig. Imidlertid trenger de foreslåtte løsningene forbedringer. Videre arbeid inkluderer å løse spørsmålet om hastighet i skriptet man opplever når man sender banepunkter til robotkontrolleren, og kravet om at den fysiske roboten skal være i sin nullposisjon før den mottar banepunkter.

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

The subject of this report is robotic offline programming. More specifically, the implementation of offline programming with the robot manipulator Yaskawa Motoman GP25-12. Robotic offline programming is a method that differs from traditional online programming in that the teach-pendant is replaced by an external computer. This entails that the work environment is moved from the factory floor to wherever; the physical robot is no longer required in the programming phase.

With Industry 4.0, robotic offline programming is more relevant than ever. The reader will be introduced to aspects of the industry supporting the implementation of robotic offline programming, as Industry 4.0, the OPC UA protocol, and different simulation software. Robot programming will also be described. A system description of the robot cell installed at Perleporten is presented, including a Python library named Moto, developed by supervisor Lars Tingelstad, allowing for communication with the robot controller. The process of developing and implementing a solution for offline programming with the Yaskawa GP25-12 is presented in detail.

1.1. Objectives

The main goal for this project, as stated above, is to implement offline programming with the robot manipualtor Yaskawa Motoman GP25-12. The report has four objectives, listed below.

- Present and assess aspects of the industry that supports the implementation of robotic offline programming
- Create a digital representation of the robot cell at Perleporten, using Visual Components Premium 4.3

- Develop a solution for offline programming that incorporates Visual Components, OPC UA, and the Moto library
- Examine the potential discrepancies between the offline programmed trajectory and the trajectory performed by the Yaskawa GP25-12.

1.2. Outline of thesis

The report is structured as follows; Chapter 2 presents the kinematics of a robot manipulator. Chapter 3 presents a aspects of the industry supporting the implementation of offline programming. Chapter 4 and 5 are specific to this project, and presents the robot cell at Perleporten, NTNU, and the process of developing and implementing the solutions, respectively. Chapter 6, 7, and 8 presents the obtained results, a discussion of the theory presented and the results, and finally a conclusion and further work.

Chapter 2.

Robot kinematics

This chapter aims to provide the reader with a basic understanding of a robot manipulator; more specifically, "Robot Kinematics" is explained. The reader will be introduced to both forward- and inverse kinematics, as well as other important concepts such as velocity kinematics, singularities, and redundancy. With an understanding of all these concepts, one is able to both plan and control the movement of a robot manipulator. Industrial robots with six degrees of freedom will be the main focus in this chapter, as such robots are the topic of the successive chapters. The chapter is gathered from the project thesis provided in the digital appendix with this report [1], while Section is new 2.3.

2.1. Rotation and translation matrices

Before introducing robot kinematics, two important matrices will be presented; the rotation matrix and the translation matrix. These matrices are essential when describing motions. To explain the concept of rotation matrices, consider first two frames; a space frame {s} and a body frame {b} which is rotated 90 degrees about the z_s axis, as shown in Figure 2.1. The orientation of the body frame {b} with respect to the space frame {s} can be described by the vectors x_b = (0,1,0), y_b = (-1,0,0) and z_b = (0,0,1). These vectors can be represented in a rotation matrix denoted R_{sb} as

$$R_{sb} = \begin{bmatrix} x_b & y_b & z_b \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.1}$$

The subscript "sb" represent the reference frame, {s} in this case, and the frame to

be transformed with respect to the reference frame, {b} in this case, respectively. The set of all 3×3 rotational matrices is called "The special orthogonal group $SO(3)$". The rotational matrices $R \in SO(3)$ are subjected to two conditions; (i) $R^{\mathrm{T}}R = I$ and (ii) det $R = 1$ [2].

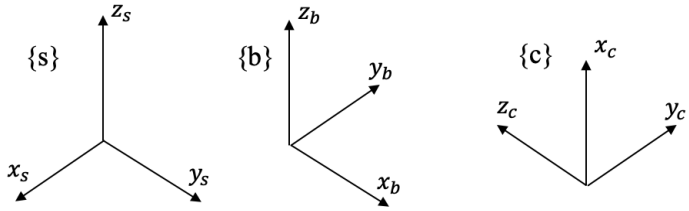


Figure 2.1.: Three frames {s}, {b}, and {c} with different orientations. Adapted from [2]

The rotation matrix in Equation (2.1) contains nine elements, but the space describing an orientation is three dimensional [2]. The elements of the rotation matrix are subjected to six constraints, summarised in condition (i) described above. This condition states that the matrix R must be orthogonal. An $n \times n$ matrix R is orthogonal if the relationship $R^{\mathrm{T}}R = I$ holds, R^{T} and I being the matrix R transposed and the identity matrix, respectively [3]. Condition (ii) contains the "special" case of the $SO(3)$ group. Whereas condition (i) ensures det $R = \pm 1$, condition (ii) states that the determinant of all matrices R must be equal to $+1$. This implies the use of the right-hand rule to determine positive and negative rotation about an axis.

Rotation matrices are commonly used for three purposes [2], the first one being to represent an orientation. Referring to Figure 2.1 and Equation (2.1), the rotation matrix R_{sb} represents the orientation of the body frame {b} with respect to the space frame {s}. A second way to take advantage of the rotation matrix is to change reference frame. Imagine a third frame, {c}, with a different orientation than {s} and {b}, as shown in Figure 2.1. Suppose one wants to express the {b} frame in {c} coordinates, as opposed to {s} coordinates. The rotation matrix R_{cb} can then be found by the following matrix multiplication $R_{cb} = R_{cs}\ R_{sb}$, due to a cancellation principle: if the second subscript of the first matrix and the first subscript of the second matrix are equal, these cancel each other.

The third way to utilize the properties of the rotation matrix is to rotate a vector or a frame. Again, referring to Figure 2.1, by studying frame {b} with respect to frame {s} one can see that the former is obtained from the latter by a rotation of 90 degrees about z_s. This can be expressed as $R_{sb} = R = \mathrm{Rot}(z, 90°)$. This rotation matrix can be used in pre- and post-multiplications to rotate any frame.

Say R_{sc} represents the orientation of frame {c} in {s} coordinates. If one pre-multiplies this matrix by the rotation operator $R = \mathrm{Rot(z, 90°)}$, frame {c} would be rotated about the z axis corresponding to the first element in the subscript of R_{sc}, z_s in this case. The rotated frame can be represented as $R_{sc'} = R\ R_{sc}$, still expressed in {s} coordinates. For post-multiplication, the rotation about the z axis would correspond to the last element in the subscript of R_{sc}; z_c, and the rotated frame becomes $R_{sc''} = R_{sc}\ R$, also still expressed in {s} coordinates [2].

The angular velocity of frame {b} expressed in {s} can be represented by a unit vector, $\hat{\omega_s}$, and the speed of rotation about it, $\dot{\theta}$. The angular velocity is given as $\omega_s = \hat{\omega_s}\ \dot{\theta}$. Further, the linear velocities of the axes of frame {b} are all a function of the angular velocity and the respective axes:

$$\dot{\mathrm{x}}_b = \omega_s \times \mathrm{x}_b, \tag{2.2}$$

$$\dot{\mathrm{y}}_b = \omega_s \times \mathrm{y}_b, \tag{2.3}$$

$$\dot{\mathrm{z}}_b = \omega_s \times \mathrm{z}_b\,. \tag{2.4}$$

An important concept called the "skew-symmetric matrix" simplifies the mathematical computations involving cross-product of vectors. Say $x = [x_1 x_2 x_3]^{\mathrm{T}} \in \mathbb{R}^3$. One can define a 3×3 skew-symmetric matrix representation of x, denoted $[x]$, which satisfies $[x] = \text{-}[x]^{\mathrm{T}}$. The set of all these 3×3 skew-symmetric matrices is called $so(3)$ [2]. The skew-symmetric matrix $[x]$ is expressed as

$$[x] = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix}. \tag{2.5}$$

The rotation matrix R usually refers to an orientation of the body frame relative to the space frame, so the subscript can be excluded. Therefore, by applying the notation of a skew-symmetric matrix explained above, a general expression for the relationship between $\dot{R}_{sb}$ and the angular velocity, ω_s, can be expressed as

$$\dot{R} = \begin{bmatrix} \dot{\mathrm{x}} & \dot{\mathrm{y}} & \dot{\mathrm{z}} \end{bmatrix} = [\omega_s]R. \tag{2.6}$$

From the equations above, one can deduce the following relations:

$$\omega_b = R^{-1}\omega_s = R^{\mathrm{T}}\omega_s \qquad [\omega_b] = R^{-1}\dot{R} = R^{\mathrm{T}}\dot{R}$$
$$\omega_s = R\omega_b \qquad [\omega_s] = \dot{R}R^{-1} = \dot{R}R^{\mathrm{T}}$$

In order to describe a motion consisting of both rotation and translation, yet another group is required; the special Euclidean group $SE(3)$. The special Euclidean group is also known as the group of homogeneous transformation matrices T in $\mathbb{R}^3$. The set of all these 4×4 transformation matrices T is called $SE(3)$. The transformation matrix T is found as

$$T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_1 \\ r_{21} & r_{22} & r_{23} & p_2 \\ r_{31} & r_{32} & r_{33} & p_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2.7}$$

where $R \in SO(3)$ and the column vector p is in $\mathbb{R}^3$ [2].

The R matrix in Equation (2.7) above is the rotation matrix presented earlier in this chapter. The vector p describes the position of the origin of the frame to be represented relative to the reference frame[2]. The last row of the transformation matrix is added for computational simplicity. Just as with the rotation matrix R, the transformation matrix T has three common applications, the first one being to represent a configuration. Referring to Figure 2.2, frame {b} is rotated by some rotation matrix R_{sb} relative to the space frame {s}, and translated by a vector p. The transformation matrix T representing frame {b} relative to frame {s} is found as

$$T_{sb} = \begin{bmatrix} R_{sb} & p_1 \\ 0 & 1 \end{bmatrix}. \tag{2.8}$$

The second application for the transformation matrix is to change reference frame. This is analogous to changing the reference frame with a rotation matrix. Referring to Figure 2.2, if we know the transformation matrices T_{sb}, calculated with R_{sb} and p_1, and T_{bc} calculated with R_{bc} and p_2, the transformation matrix representing frame {c} relative to frame {s} can be found as $T_{sc} = T_{sb}\, T_{bc}$.

The third way to utilize the transformation matrix is to displace a vector or a frame. Any given configuration can be achieved by first translating and then

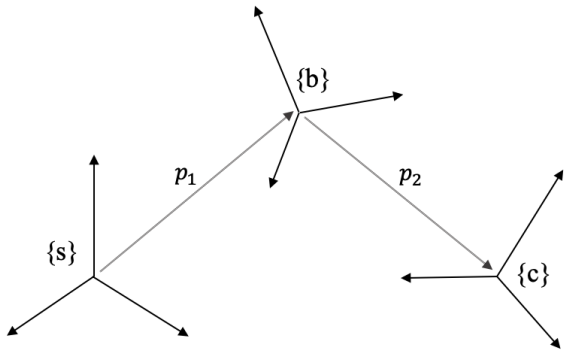


Figure 2.2.: Frames {s}, {b}, and {c} with different positions and orientations relative to each other. Adapted from [2]

rotating a frame. Mathematically, this can be expressed as

$$T = \text{Trans}(p)\,\text{Rot}(\hat{\omega}, \theta) = \begin{bmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} & & & 0 \\ & e^{[\hat{\omega}]\theta} & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.9}$$

Referring to frames {s} and {b} in Figure 2.2, the transformation matrix representing frame {b} relative to frame {s} is given as T_{sb}, and the displacement can be found by either pre- or post-multiplying the transformation matrix T_{sb} with T (2.9). Whether the variables p and $\hat{\omega}$ in Equation (2.9) are expressed in {s} or {b} depends on the order of the multiplication. When pre-multiplying, TT_{sb}, both p and $\hat{\omega}$ are expressed in {s}. This implies a rotation about the axis represented by $\hat{\omega}$, followed by a translation defined by p, both in the {s} frame. When post-multiplying, $T_{sb}T$, both p and $\hat{\omega}$ are expressed in {b}. This implies a translation defined by p followed by a rotation about the axis represented by $\hat{\omega}$, both in the {b} frame.

The 4×4 rotation matrix $\text{Rot}(\hat{\omega}, \theta)$ in Equation (2.9) contains an element $e^{[\hat{\omega}]\theta} \in SO(3)$. This is the so-called "matrix exponential" which is defined as follows:

$$\text{Rot}(\hat{\omega}, \theta) = e^{[\hat{\omega}]\theta} = I + \sin\theta[\hat{\omega}] + (1 - \cos\theta)[\hat{\omega}]^2 \in SO(3). \tag{2.10}$$

An analogous representation for the transformation matrix in $SE(3)$ is given as

$$T = e^{[\mathcal{S}]\theta} = \begin{bmatrix} e^{[\omega]\theta} & (I\theta + (1 - \cos\theta[\omega]) + (1 - \sin\theta)[\omega]^2)v \\ 0 & 1 \end{bmatrix}, \tag{2.11}$$

for $||\omega|| = 1$, and for $\omega = 0$ and $||v|| = 1$ the expression becomes

$$T = e^{[\mathcal{S}]\theta} = \begin{bmatrix} I & v\theta \\ 0 & 1 \end{bmatrix}. \tag{2.12}$$

2.2. Robot kinematics

A robot manipulator consists of n joints connected by $n+1$ links, where actuators provide motive power allowing the links to move. The number of movable joints determines the robots degree of freedom, DOF, where at least six DOF is required in order to fully describe an objects position in space [4]. Further, the number of joints determines the dimension of the configuration space, which contains all possible configurations of the robot. A robot with six DOF, implies a configuration space of dimension six. The task-, also called cartesian space of a robot, is where the task to be performed is expressed, and its dimension is determined by the number of variables required to describe the position of the end-effector. For a six DOF robot, the position and orientation of the end-effector is described by the coordinate system {x,y,z} and the rotation around each of the axes, often referred to as roll, pitch, and yaw. This implies a task space of dimension six. In fact, the dimension of the task space can not be higher than six, whereas the dimension of the configuration space can. However, if the dimension of the configuration space is greater than the dimension of the task space, the robot is said to be redundant. Redundancy is explained in Section 2.8. Whereas the task space expresses the tasks of a robot´s end-effector, the workspace specifies all the possible configurations of the end-effector [2].

Robot kinematics is a crucial tool in both understanding and controlling the motion of a robot. The kinematic model is used to describe the robot´s motion, excluding the forces required to achieve these motions [5]. Within kinematics, one differs between forwards- and inverse kinematics. While forward kinematics uses known information about the joint angles to calculate the position of the end-effector, inverse kinematics calculates the joint angles based on the desired position of the end-effector. A simple sketch is shown in Figure 2.3 to illustrate the concept. Forward kinematics problems are quite straight-forward to calculate, while inverse kinematics offers much more complex and time-consuming calculations.

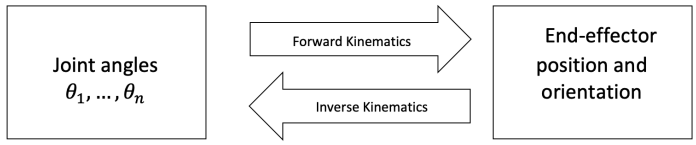


Figure 2.3.: Simple illustration comparing forward kinematics and inverse kinematics. Adapted from [6]

2.3. Path and trajectory

The words´ path and trajectory are important concepts in robotics, used to describe the movement of a robot. Even though they both refer to the same concept, movement, they describe the movement differently and must therefore be used correctly.

When talking about the path of a robot, one refers to a movement from A to B defined by a set of points. It is a spatial construct that describes the movement [7]. Not described by the path is the speed at which the robot is to execute its motion from A to B; there is no notion of time. A trajectory describes the path and the speed at which the robot is to move from A to B. There are different ways of generating trajectories.

One method is to use a so-called quintic polynomial with six coefficients, defined as

$$S(t) = At^5 + Bt^4 + Ct^3 + Dt^2 + Et + F\,, \qquad t \in [0, T]. \tag{2.13}$$

This permits one to specify six constraints; the initial and final position, denoted s_0 and s_T, the initial and final velocity, $\dot{s}_0$ and $\dot{s}_T$, and the initial and final acceleration, $\ddot{s}_0$ and $\ddot{s}_T$. However, one problem with this method is that motor performance is wasted as the peak velocity is almost double the average velocity.

Another method is the trapezoidal trajectory. This is more efficient than the former and is characterized by an acceleration phase, followed by a constant velocity, before finally a deceleration phase. One small problem with this trajectory generation method is that the acceleration is discontinuous, however, it is widely used in robotics.

The path of a robot can be simply a straight line, or many consecutive straight lines, resulting in a non-linear slope [8]. Each of these linear segments are defined by so-called “via points” or “way points”. For the robot to follow such a path with complete precision, it would have to come to a complete stop at each via point.

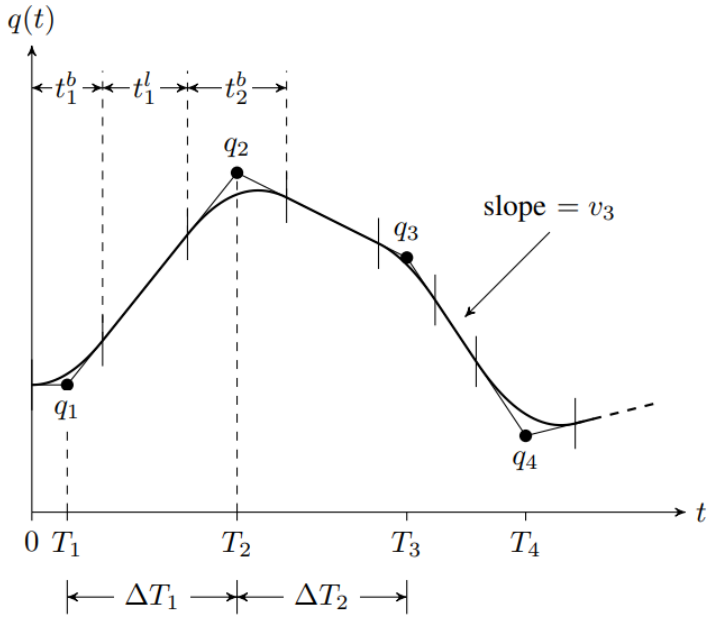


Figure 2.4.: Non-linear slope defined by four way-points [8]

This is due to the fact that robots have nonzero mass and also the forces that can be applied are finite, which means that the robot is incapable of changing its direction of motion instantaneously. In order to achieve continuity, so-called parabolic blends can be added. These blends are shown in Figure 2.4 as continuous curved segments near each via point. The time t_2^b "containing" via point q_2 is called the acceleration period, in which there is a transition from the velocity from the previous via point, q_1, to the velocity towards the following via point, q_3. As can be seen in the figure, the continuous slope is not actually getting to the via points. There is a trade-off between accuracy and acceleration [7]. A high acceleration leads to a small t_2^b, which allows for the trajectory to get close to the via point. A low acceleration leads to a high t_2^b, but a larger position error from the via point.

Another way is through interpolation:

$$x(s) = (1 - s)x_0 + sx_1\,, \qquad s \in [0, 1]. \tag{2.14}$$

This method takes in a scalar value, s, that varies between 0 and 1, where x(0) gives the initial value and x(1) gives the final value. Here, x can be a vector or a smooth function of time if s is that. However, this method is not applicable for

rotation matrices.

Rotation matrices are orthogonal matrices, explained in Section 2.1. Adding two orthogonal matrices does not result in an orthogonal matrix [7]. However, since x can be a vector, one can convert the rotation matrix to a vector Γ,

$$\Gamma(s) = (1-s)\Gamma_0 + s\Gamma_1\,, \qquad s \in [0,1], \tag{2.15}$$

containing a set of angles, represented as Euler angles or roll-pitch-yaw angles. However, interpolating angles requires one to take into account the direction, as there will be two ways to get from A to B.

2.4. Forward kinematics

Using known information about the joint angles of the robot to calculate the position and orientation of the end-effector is called forward kinematics. As the variables in the equations are known, forward kinematics always yields a solution, and the solution is unique. There are two main methods for calculating the forward kinematics for a given robot manipulator; Denavit-Hartenberg and Product of Exponentials [2].

2.4.1. Denavit-Hartenberg

There exists two versions of the Denavit-Hartenberg method, the one described in this section is the so-called "Modified Denavit-Hartenberg" [2]. This method describes each joint and link of the robot with four parameters; ϕ_i, d_i, a_{i-1}, and α_{i-1}, for i = 1 to n, n being the number of joints. The parameters describe the joint angle, link offset, link length, and link twist, respectively. The method then involves attaching a coordinate frame $\{\mathrm{x}_0, \mathrm{y}_0, \mathrm{z}_0\}$ to $\{\mathrm{x}_n, \mathrm{y}_n, \mathrm{z}_n\}$ to each joint, where the z_i axis points in the direction of the rotational/linear movement [9]. Each link transform is represented by a homogeneous transformation matrix T_i^{i-1} as

$$\begin{aligned} \mathrm{T}_i^{i-1} &= \mathrm{Rot}(\hat{\mathrm{x}}, \alpha_{i-1})\,\mathrm{Trans}(\hat{\mathrm{x}}, a_{i-1})\,\mathrm{Trans}(\hat{\mathrm{z}}, d_i)\,\mathrm{Rot}(\hat{\mathrm{z}}, \phi_i) \\ &= \begin{bmatrix} \cos\phi_i & -\sin\phi_i & 0 & a_{i-1} \\ \sin\phi_i\ \cos\alpha_{i-1} & \cos\phi_i\ \cos\alpha_{i-1} & -\sin\alpha_{i-1} & -d_i\ \sin\alpha_{i-1} \\ \sin\phi_i\ \sin\alpha_{i-1} & \cos\phi_i\ \sin\alpha_{i-1} & \cos\alpha_{i-1} & d_i\ \cos\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \end{aligned} \tag{2.16}$$

where four elementary transformations are made; a rotation around z, a translation along z, a translation along x and a rotation around x [2].

As previously stated, only four parameters are required to express the forward kinematics when applying the Denavit-Hartenberg method. With the Product of Exponentials, six parameters are required in order to describe the displacement in terms of orientation and position. The Denavit-Hartenberg method introduces two constraints on the placement of each coordinate frame $\{\mathrm{x}_i, \mathrm{y}_i, \mathrm{z}_i\}$, which results in unique values for ϕ_i, d_i, a_{i-1}, and α_{i-1}. The first constraint states that axis x_i intersects axis z_{i-1} and the second that axis x_i is perpendicular to axis z_{i-1}. Each of the four parameters is then found by analyzing link i with respect to link $i-1$. Usually, the parameters are placed in a table, as shown in Table 2.1 where $i = 1$ to n, n being the number of joints. The solution to the forward kinematics problem is found by multiplying the n transformation matrices as

$$\mathrm{T}^{\mathrm{base}}_{\mathrm{EndEffector}} = \mathrm{T}^0_1 ... \mathrm{T}^{i-1}_i \,. \tag{2.17}$$

This gives the position and orientation of the end-effector frame with respect to the base frame [6].

Table 2.1.: Example set-up of table with Denavit-Hartenberg parameters, where subscript x is some value between zero and n

i	α_{i-1}	a_{i-1}	d_i	ϕ_{i-1}
1	0	0	0	θ_1
⋮	⋮	⋮	⋮	⋮
i=n	$\pm$ 90°	$\mathrm{L}_{i-\mathrm{x}}$	0	θ_n

2.4.2. Product of Exponentials

While the Denavit-Hartenberg method is quite straight-forward, it can be time-consuming. When calculating the forward kinematics using the Product of Exponentials, PoE, one simply has to define two frames; a fixed base frame {s} and an end-effector frame {b}. It is, however, convenient to assign a frame to each of the robot´s n joints, where the z_n axis of each frame points in the direction of positive rotation [2]. The end-effector frame is described by a 4×4 matrix, M, which is determined when the robot is at its zero position, meaning all joint angles are equal to zero. The first three columns of M are determined by comparing $\{\mathrm{x}_b, \mathrm{y}_b, \mathrm{z}_b\}$ to the base frame $\{\mathrm{x}_s, \mathrm{y}_s, \mathrm{z}_s\}$. If for example x_b points in the negative direction of z_s, the first column of M would be $[0 \quad 0 \quad -1 \quad 0]^T$. The fourth and last column of M is set by evaluating the respective link lengths required to move the base

frame $\{x_s, y_s, z_s\}$ to the end-effector frame $\{x_b, y_b, z_b\}$. The forward kinematics with the PoE formula is calculated as

$$T(\theta) = e^{[\mathcal{S}_1]\theta_1} \dots e^{[\mathcal{S}_{n-1}]\theta_{n-1}} \, e^{[\mathcal{S}_n]\theta_n} M, \tag{2.18}$$

where i = n, n being the number of joints.

As can be seen from Equation (2.18), the PoE formula requires one to define so-called "screw axes", $\mathcal{S}_i$. Each screw axis is a column vector with six elements, determined by ω_i and v_i, resulting in the expression: $\mathcal{S}_i = (\omega_i, v_i)$, for $i = 1$ to n, n being the number of joints [2]. The vector ω_i contains three elements and describes the i^{th} joint rotation with respect to the base frame. If the axis of rotation for join i points in the direction of -y_s, then ω_i = (0,-1,0). The vector v_i is found by the cross-product between -ω_i and q_i, where q_i is a vector with three elements describing the i^{th} joint translation with respect to the base frame. If for example link 2 has length L_1 in positive x_s direction, while the translation in y_s and z_s is zero, then q_2 = (L_1, 0, 0). The screw axes $\mathcal{S}_i$ in Equation (2.18) are expressed in matrix exponential form, which are found as shown in Equation (2.12). The solution to the forward kinematics problem for any given joint angle θ_i is given by Equation (2.18) with the respective screw matrices in exponential form $e^{[\mathcal{S}_i]}$ and the M matrix [2].

The above derived forward kinematics with the PoE is called the Space form, implying that the screw axes are represented in the base frame. Another representation of the PoE is the Body form, where the screw axes are represented in the end-effector frame; $\mathcal{B}_i = (\omega_i, v_i)$, for $i = 1$ to n, n being the number of joints [2]. With the body form, the M matrix representing the end-effector configuration at zero position is found by following the same procedure as with the space form above. The same applies for the respective screw axes expressed in the end-effector frame, but the vectors ω_i and v_i are determined with respect to the end-effector frame. If the axis of rotation of joint i points in the direction of y_b, then ω_i = (0,1,0). The vector v_i results from the cross-product of -ω_i and q_i, just as with the space form. The solution to the forward kinematics calculated in body form is given as

$$T(\theta) = M e^{[\mathcal{B}_1]\theta_1} \dots e^{[\mathcal{B}_{n-1}]\theta_{n-1}} \, e^{[\mathcal{B}_n]\theta_n}, \tag{2.19}$$

where the screw axes expressed in exponential matrix form, $e^{[\mathcal{B}_i]}$ are found as in Equation 2.12.

2.5. Inverse kinematics

Inverse kinematics transforms the position and orientation of the end-effector from the Cartesian-, or task space to the joint space, which is represented by the joint angles. Referring to Section 2.4.2, the solution to the forward kinematics problem using PoE was given as Equation (2.18). With inverse kinematics, one seeks to obtain the angles required to reach the desired end-effector position. The inverse kinematics problem can be expressed mathematically as

$$X = T(\theta) \tag{2.20}$$

where X represents the desired end-effector configuration [2]. As opposed to forward kinematics, which always yields one unique solution, inverse kinematics can give several valid joint angles for one and the same end-effector position. This means that there exist multiple configurations of the robot manipulator while the end-effector position remains the same. However, the desired solution to the inverse kinematics problem is the one that minimizes the joint motion while ensuring that the robot does not collide with itself [5]. As with forward kinematics, inverse kinematics problems can be solved using different methods. However, in contrast to forward kinematics, the choice of method does not depend on preference but rather the ability to achieve a solution. Further, regardless of the choice of method, one might not be able to find any solution to the inverse kinematics problem. The robot´s workspace was introduced at the beginning of this chapter as a space containing all reachable points for the robot manipulator. For inverse kinematics problems where X, the end-effector configuration, lies outside the robot´s workspace, there will not exists any solution [10].

2.5.1. Analytical inverse kinematics

Solving an inverse kinematics problem analytically is mathematically challenging, and the complexity increases with the number of joints. Ultimately the analytical inverse kinematics problem might become unsolvable. There is no “one method” when solving an analytical inverse kinematics problem; the mathematical computations depends on the robot manipulator in question. The general approach can be explained as follows: First, one needs to obtain the position of the end-effector by solving the forward kinematics problem. From the transformation matrix representing the pose of the end-effector, one extracts equations and solve these with respect to the joint angles. The equations obtained from the forward kinematics are nonlinear. Solving for the angles to obtain the inverse kinematics may therefore be difficult, with increasing complexity as the number of joints increase.

The number of joints influences the computational challenge, but also their ge-

ometric arrangement [11]. If the robot in question has a spherical wrist, the analytical inverse kinematics problem can be solved by decoupling the problem into inverse position and inverse orientation. A spherical wrist is a term describing a robot manipulator whose three wrist joints intersect at a single point. Most 6 DOF industrial robots have such spherical wrists [12]. Decoupling the inverse kinematics problem simplifies the computation [2]. The first three angles θ_1, θ_2, and θ_3 are obtained by the process explained above. With these angles derived, one can solve the inverse orientation by modifying Equation (2.18) into

$$e^{[\mathcal{S}_4]\theta_4} e^{[\mathcal{S}_5]\theta_5} e^{[\mathcal{S}_6]\theta_6} = e^{-[\mathcal{S}_3]\theta_3} e^{-[\mathcal{S}_2]\theta_2} e^{-[\mathcal{S}_1]\theta_1} X M^{-1}. \tag{2.21}$$

Even though it is possible to obtain a solution to the analytical inverse kinematics problem for a six DOF industrial robot, the method is demanding and time-consuming. An alternative approach to solving the inverse kinematics for a given robot is a numerical approach.

2.5.2. Numerical inverse kinematics

Using numerical methods to solve the inverse kinematics problem is usually applied to robot manipulators where an analytical solution is unavailable. As stated in Section 2.4, one can always find the forward kinematics solution to any given robot manipulator. If one knows the desired end-effector configuration as well, one can simply adjust the angles so that the solution to the forward kinematics problem matches the desired end-effector configuration. This is usually achieved through an iterative process, such as the Newton-Raphson method [2]. This method is based on making an initial guess of the joint angles as well as determining two positive error allowance; one for the orientation and one for the linear position of the end-effector. The resulting angles must give an end-effector configuration satisfying the allowed errors. The initial guess of the joint angles must be sufficiently close to the solution for the iterations to converge, i.e., providing a solution.

2.6. Velocity kinematics

The previous sections of this chapter have been concerned with the position and orientation of the end-effector, expressed as forward- and inverse kinematics problems. However, the motion required to achieve these configurations involves translational and rotational movement. This is velocity kinematics.

The linear and angular velocity of an end-effector can be described by two components, ω and v, both column vectors with three elements. This results in a

column vector with six elements, denoted $\mathcal{V}$. The column vector $\mathcal{V}$, expressed as

$$\mathcal{V} = \begin{bmatrix} \omega \\ v \end{bmatrix} \in \mathbb{R}^6, \tag{2.22}$$

describing the velocity of the end-effector is called the twist, or spatial velocity [2]. The twist $\mathcal{V}$ is determined by following the same procedure as with the screw axes in Section 2.4.2, the difference being that the screw axes in Section 2.4.2 were determined while the robot was at its zero position, implying $\theta = 0$. For the twist, $\mathcal{V}$, the screw axes depend on θ.

Further, the matrix representation of a twist can be written as

$$[\mathcal{V}] = \begin{bmatrix} [\omega] & v \\ 0 & 0 \end{bmatrix} \in se(3). \tag{2.23}$$

Note that the above equations, (2.22) and (2.23), are written in general form. These expressions are valid for representing the twist in both space form and body form, called spatial twist and body twist, respectively. It can be shown that the relationship between the spatial twist and the body twist can be written as

$$[\mathcal{V}_b] = T^{-1}\dot{T} = T^{-1}[\mathcal{V}_s]T, \tag{2.24}$$

and

$$[\mathcal{V}_s] = T[\mathcal{V}_b]T^{-1}, \tag{2.25}$$

where the matrix T is the transformation matrix derived in Section 2.1. Writing out Equation (2.25) yields the following relationship between the spatial twist and the body twist

$$\begin{bmatrix} \omega_s \\ v_s \end{bmatrix} = \begin{bmatrix} R & 0 \\ [p]R & R \end{bmatrix} \begin{bmatrix} \omega_b \\ v_b \end{bmatrix}. \tag{2.26}$$

The 2×2 matrix in Equation (2.26) pre-multiplying $\mathcal{V}_b$ is called the adjoint representation of T, denoted $[\mathrm{Ad}_T]$. This matrix is useful when changing between frames, where subscript T denotes the transformation matrix expressed in space form, T_{sb}, or body from, T_{bs}.

An important concept within robot kinematics must be addressed: the Jacobian matrix $J(\theta)$. The Jacobian is a matrix that provides a relationship between the joint velocities, $\dot{\theta}$, and the tip velocity vector, v_{tip} [2]. The velocity vector v_{tip} can be expressed in several ways; in the following section, the end-effector velocity will be represented by the twist $\mathcal{V}$. The relationship between the joint velocities and the twist can be expressed as

$$\mathcal{V} = J(\theta)\dot{\theta}. \tag{2.27}$$

For a robot manipulator the Jacobian is given as $J(\theta) = [J_1(\theta) \; .. \; J_n(\theta)]$, where n is the number of joints. This implies that the number of columns in the Jacobian is equal to the number of joints in the robot manipulator. Further, $J_i(\theta)$ is the twist $\mathcal{V}_i$ when the corresponding $\dot{\theta}_i = 1$ and all other joint velocities are equal to zero.

As with the screw axis in Section 2.4.2, one can define a space Jacobian, $J_s(\theta)$, and a body Jacobian, $J_b(\theta)$. For the former, the elements of $J_s i(\theta)$ are set by the respective screw axis expressed in {s} frame. For the latter, the elements of $J_b i(\theta)$ are set by the respective screw axis expressed in {b} frame [2].

As mentioned above, the screw axes, $\mathcal{S}_i$ and $\mathcal{B}_i$, of a robot are determined while the robot is at its zero-position, implying all joint angles are equal to zero. However, the Jacobian $J(\theta)$ is defined for any arbitrary value of θ. The space Jacobian $J_s(\theta)$ for a robot manipulator with n joints is defined as

$$\mathcal{V}_s = J_s(\theta)\dot{\theta} = [J_{s1} \; J_{s2}(\theta) \; ... \; J_{sn}(\theta)]\dot{\theta}, \tag{2.28}$$

where $J_{s1} = \mathcal{S}_1$, and

$$J_{si}(\theta) = [\text{Ad}_{e^{[\mathcal{S}_1]\theta_1 ... e^{[\mathcal{S}_{i-1}]\theta_{i-1}}}}]\mathcal{S}_i, i = 2, ..., n. \tag{2.29}$$

Here, the adjoint mapping Ad_{T} is used to represent the new screw axis with some arbitrary values of the joint angles, and the transformation matrix T is given as $\text{e}^{[\mathcal{S}_1]\theta_1} ... \text{e}^{[\mathcal{S}_{i-1}]\theta_{i-1}}\mathcal{S}_i$. The same reasoning applies for the body Jacobian $J_b(\theta)$ which, for a robot manipulator with n joints, is defined as

$$\mathcal{V}_b = J_b(\theta)\dot{\theta} = [J_{b1}(\theta) \; ... \; J_{bn-1}(\theta)J_{bn}]\dot{\theta}, \tag{2.30}$$

where $J_{bn} = \mathcal{B}_n$, and

$$J_{bi}(\theta) = [\mathrm{Ad}_{e^{-[\mathcal{B}_n]\theta_n}\ldots e^{-[\mathcal{B}_{i+1}]\theta_{i+1}}}]\mathcal{B}_i, i = 1, ..., n-1. \tag{2.31}$$

2.7. Singularities

Singularities, when talking about robots, refers to a configuration of the robot manipulator which prohibits the end-effector from moving in certain directions [2]. This implies that when a robot manipulator is at a singularity, it loses one or more of its degrees of freedom. Any robot manipulator with six degrees of freedom will have such singularities, at which its mobility will be limited. However, the complexity and type of singularity depend on the type of joints, the number of joints, and how the joints are configured [11].

Referring to Section 2.6, the Jacobian gives information about a robot manipulators singularities. For a six DOF robot, the resulting Jacobian matrix will be 6×6. Singularities for the six DOF robot will be where the Jacobian is not of maximum rank.

2.8. Kinematically redundant

A kinematically redundant robot manipulator is a manipulator that consists of more than six joints, i.e., $n > 6$. In the introduction to this chapter, it was stated that at least six DOF was required in order to fully describe the end-effector´s configuration in space. Further, the task space of any given robot manipulator can not have a dimension larger than six. For a robot with $6 + n$ DOF, the n successive joints are excess in terms of describing the end-effector configuration; they are redundant. However, the excess joints are useful in obstacle avoidance and for optimizing objective functions [2].

Chapter 3.

Aspects of the industry supporting robotic offline programming

This chapter aims to provide the reader with an understanding of some aspects that essentially enable robotic offline programming. The first section will introduce Industry 4.0; what it is, and how it affects the industry today. The communication protocols OPC Classic and OPC Unified Architecture will be presented as a part of this section. Next, robot programming will be presented, where part of the section has been gathered from the project thesis provided in the digital appendix with this report [1]. Online programming will be described shortly before introducing offline programming, which is a newer method for programming robots that is highly relevant for the mindset of Industry 4.0. The last section will present a selection of simulation software available in the industry today.

3.1. Industry 4.0

The manufacturing industry of today has been formed by four important eras that have revolutionized the industry during the past hundred years [13]. The first industrial revolution, the age of mechanical production, is recognized by the invention and implementation of steam power and water power in manufacturing [14]. The second industrial revolution, the age of science and mass-production, concerned the invention of mass-production assembly lines and electricity. The third revolution, the digital age, introduced electronics, I.T systems and enabled automation in production. The fourth industrial revolution, the technological age, is often referred to as Industry 4.0 and represents the ongoing shift from the third industrial revolution. Industry 4.0 is possible due to so-called cyber-physical sys-

tems, the Internet of Things and the Internet of Systems, and advances from the third industrial revolution by connecting people, machines, and physical assets that ultimately, by incorporating data and machine learning, can be able to make decisions without human intervention [13].

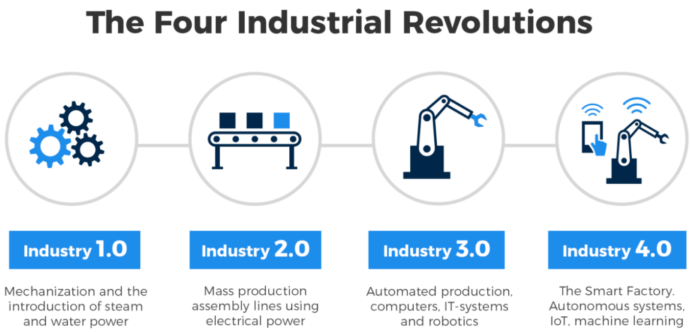


Figure 3.1.: The four industrial revolutions [15]

As Industry 4.0 can be said to be an extension of Industry 3.0, many of the components that make out Industry 4.0 already exists [16]. It is how these components are digitally connected and can both generate and share information, illustrated in Figure 3.2, that elevates Industry 4.0 from the former. The machines from Industry 3.0 become smart machines in Industry 4.0 as they are fueled with data, which contributes to a more efficient, productive and less wasteful production.

Figure 3.2.: Illustration of Industry 4.0 [16]

Companies have been collecting process data for a while; however, many only used this data for logging purposes and not to actually improve their operations [14]. Today, customers have more options and higher demands which requires manufacturers to adapt more rapidly. It is becoming increasingly difficult to release new products to the market as the competition is high due to the number of products released at a high rate and the number of available options. Further, as products

are becoming more complicated, it is necessary for more aspects of the business to be working together. There is a shift from the mass production seen in Industry 3.0 to a more customized production as customer demands personalized products.

Features of Industry 4.0 include, amongst others, the following; virtualization, real-time capability, decentralization and interoperability. Virtualization regards creating a digital representation of the production plant, which, by receiving and outputting data with the help of sensors, reflects the status of the physical plant. Such a digital representation is often called a Digital Twin. Real-time capability refers to the process of collecting and analyzing data and taking immediate action based on the information. Digital twins operate in real-time. Decentralization concerns cyber-physical systems, hardware with software embedded, capable of making their own decisions and thereby enable local production. Interoperability is about how systems can communicate with one another, which proved to be challenging [17].

The three previous revolutions have contributed to well established infrastructure and systems, which have evolved and, during the last 30 years, become more and more automated. One problem lies within how to make these existing systems interoperable, which motivates the introduction of an important protocol named Open Protocol Communications Unified Architecture, OPC UA. OPC UA is a well-renowned communication protocol with its maybe most important feature being that it is platform independent.

3.1.1. Before Open Protocol Communications

The manufacturing industry grew extensively with the third industrial revolution, which introduced the need for data from the factory floor. Motivated by this, automation vendors Fisher-Rosemount, Intellution, Opto 22 and Rockwell Software established the OPC Foundation in 1996 [18]. The OPC Foundation was founded with the aim of solving problems with interoperability, compliance, validation and certification [17].

Before the OPC Foundation was established, there did not exist any good standard for communication between industrial control devices and computer applications designed to use real-time data from these controllers. Some attempts were made, e.g. Dynamic Data Exchange, DDE, from Windows 3.0 [19]. This standard mechanism made it possible for multiple applications to exchange data at run-time. However, it proved to be fragile, have limited bandwidth, and lacked support across a network. Several attempts were made to fix these limitations, where InTouchTM SCADA software might have been the most successful. However, this and others, like AdvanceDDETM, were all proprietary, implying that they required payments to their investors, thereby limiting their chances of becoming

an industry standard.

OLE 2.0 was released in 1992 and, with time, proved to be the replacement of DDE [19]. A group named WinSEM expressed interest in the possibilities with OLE techniques for data exchange between applications in real-time. Especially SCADA vendors were looking into the possibilities of standardizing the interface between SCADA and the device drivers acquiring the data. The most promising, however in hindsight very simplistic, solution was proposed by US Data in 1995. The progress however, was slow. It was agreed that in order to deliver a standard within reasonable time, a smaller group was necessary; the OPC Task Force was established.

3.1.2. OPC Classic

OPC is based on Client/Server communication, illustrated in Figure 3.3 [20]. The interactions between the client and server are controlled by the client and can be summarized as follows:

- The client initiates communication with server
- The client manages the behaviour of server
- The server waits for incoming requests from client
- The server only performs actions as instructed by the client

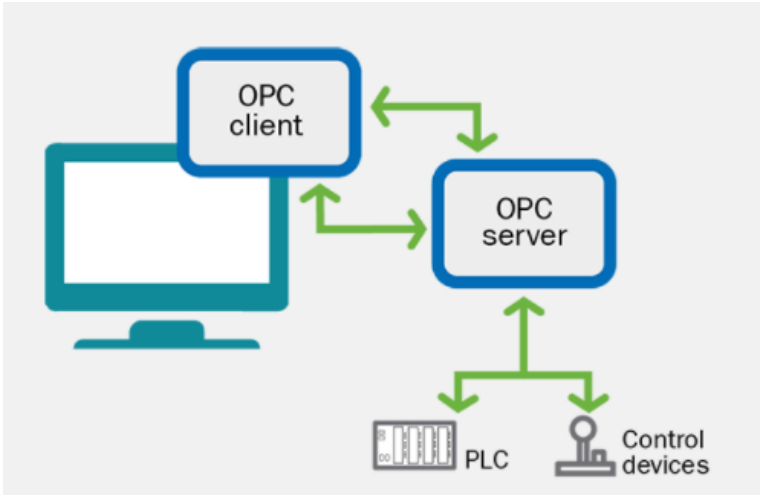


Figure 3.3.: OPC client-server [20]

OPC Classic refers to the original OPC specification [20]. It consists of five protocols that are independent of one another. They are not compatible; they

have their own commands for, e.g., read and write, which only affect the protocol in question.

OPC Classic started with a simple protocol named OPC Data Access, OPC DA. This protocol establishes communication between the control system and the shop floor by retrieving data from the former and sending it to the latter. The data is recognized by a "Value", which is the data itself, and a "Name". Further, a "Timestamp" is assigned to the data point containing the time of read, and lastly, "Quality" reflects whether the data is valid or not. The second protocol released by the OPC Foundation was OPC Alarm & Events, OPC AE. This protocol differs from DA in that events do not have current value, and therefore no Value, Name or Quality. The protocol does, however, have Timestamp. Following OPC EA was the OPC Historical Analysis, OPC HDA. The HDA protocol differs from the others with its support for recordsets of data for one or several data points.

While OPC Classic enables the exchange of process data, alarms and historical data, it does not operate at the machine-to-machine communication layer. Other issues with OPC Classic include its dependency to Microsoft, making it incompatible with other platforms, its vulnerability to attacks, and its inconsistencies in data and inadequate data models [17].

3.1.3. OPC Unified Architecture

With Industry 4.0, the need for standardized data connectivity and interoperability grew. As OPC Classic was developed with Windows-specific technology such as OLE and DCOM [21], it was incompatible with other platforms such as Apple and Linux, which proved to be a big constraint. Further, as explained in Section 3.1.2, the specifications of OPC Classic are not compatible with one another. The OPC UA protocol was released in 2008 and incorporates all the functionalities provided by OPC Classic into one specification. OPC UA is used in industrial automation for communication between equipment and systems for both data collection and control. The OPC UA standard was designed to fulfil the following goals [22]:

- Functional equivalent to OPC Classic
- Platform independence ensuring incorporation with any operating system and in any environment
- Secure communication with regards to encryption, authentication, and auditing
- Extensible design allowing for adding new features without affecting the exciting application

- Comprehensive information modelling compatible with the requirements of Industry 4.0

The OPC UA information model framework, illustrated in Figure 3.4, is probably the most important feature of the standard. The information model has object-oriented capabilities, allowing for multi-layered structures to be modelled, essentially turning data into information [23]. Generally, information represented in a computer system is stored in a so-called address space. A variable is created and given a unique identity, typically a string.

Instead of having one unique identifier for one given variable, the OPC UA protocol uses one identifier to represent a group of variables [24]. This identifier is called a node. Each node has a specific node ID composed of three components. The node ID is defined automatically by the server and is unique. An example of a node ID can be "ns=3,s=Counter". Here, "ns" is the namespace index. The number 3, in this example, is the identifier, and "s" is the identifier type, which in this example is a string.

As stated above, the information model has object-oriented capabilities. The node is the basic building block of the information model. However, for complex information to be processed, the node has to be able to exhibit different shapes. This functionality is why the information model is compared to object-oriented programming, as different classes of nodes are defined that all inherit from the same basic node. Examples of OPC UA node classes are Variable Node Class, Method Node Class and Object Node Class.

OPC UA provides four basic mechanisms for retrieving data. However, these can be easily modified and extended in order to retrieve specific information [22].

- Data- and event notifications
- Read/write both current and historical data
- Execution of methods
- Features for locating instances and their semantic

There are two ways of utilizing the information modelling framework with OPC UA; client/server communication and PubSub, Publish-Subscribe. The major difference between the two is that whereas the former is somewhat limited with its one-to-one or one-to-many feature, the PubSub enables many-to-many configurations [25]. A message broker, for example, MQTT, receives information from publishers. Subscribers may subscribe to specific topics of interest published on the message broker and only be notified when these topics update. A client/server communication is based on an exchange of requests and responses, where a server can handle requests from several clients.

In order to establish communication between the OPC UA server and client, an endpoint URL must be defined, consisting of three informative components. The OPC UA standard is compatible with several transport protocols, such as SOAP/HTTP (Simple Object Access Protocol/Hypertext Transfer Protocol), HTTPS (Hypertext Transfer Protocol Secure), and UA TCP (Transmission Control Protocol) [26]. The client must support one of these, and the transport protocol used is defined in the endpoint URL as the first of the three components. Further, the IP address of the device hosting the server must be included, as well as a port number. The endpoint URL will have the form "opc.tcp://<IPaddress>:<port>".

In 2016, the OPC Foundation announced that the OPC UA technology was made open-source available[27]. The reason for doing so was based on the OPC Foundation´s vision of becoming the standard for communication in the industry. By making the technology open-source, the OPC Foundation envisioned becoming the solution for the "Internet of everything". The repository can be found on the website Github http://github.com/opcfoundation.

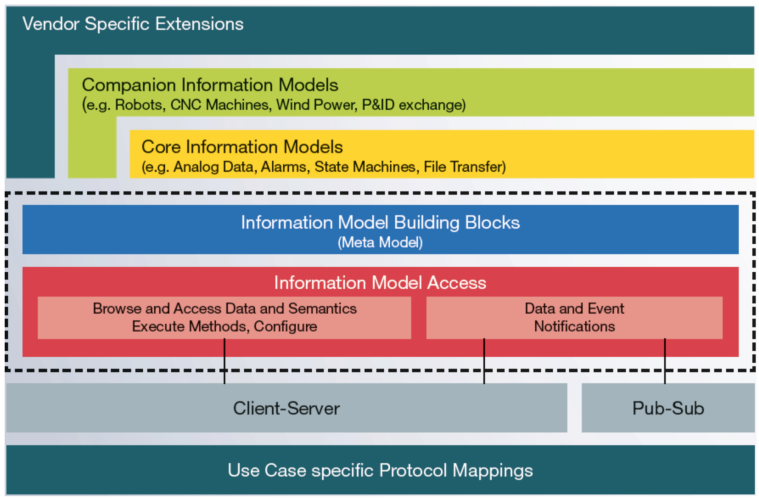


Figure 3.4.: Information model structure [22]

3.2. Robot Programming

There are two main methods for programming a robot's trajectory in the industry today; online and offline. With "traditional" online programming, a so-called teach-pendant is used to program the robot's trajectory. The teach-pendant serves as an interface between the operator and the robot and thereby requires the operator to be in, or near, the robot cell. With online programming, the operator can move the robot to the desired positions with the buttons on the pendant and

save each point continuously. This implies that production must be stopped while the operator programs the robot, which is one of the main drawbacks with online programming [28]

The teach-pendant is a handheld device the operator can use to, amongst others, program the robot, start and stop the program, and retrieve information about the robot from the digital display. The teach-pendant is usually equipped with different modes, each with its own set of constraints [29]. In teach mode, the operator prepares and teaches the robot the job to be done, as explained above. Play mode allows the operator to playback the trajectory programmed in teach mode. Both teach- and play mode have velocity constraints to restrict the robot from moving at high velocities, ensuring safety while constructing and adjusting the program. The last mode is called remote mode. While in remote mode, the program can be executed at full speed. However, remote mode should never be used before the entire program has been verified in play mode.

The path of the robot is programmed point by point with typical commands such as "MoveJ", "MoveL", and "MoveC". The chosen command depends on the required movement; "MoveJ" will make the robot move from A to B on all axes simultaneously, implying that the path from point A to point B will depend on the configuration of the robot at point "A-1". "MoveL" is used when the movement from A to B is important, the end-effector will move linearly from A to B. "MoveC" is used for circular movement. Figure 3.5 shows an example of a simple path with five points, and Table 3.1 shows the instructions required to achieve this path with the teach-pendant.

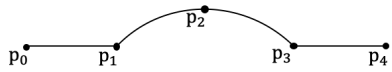


Figure 3.5.: Example of simple path with half circle movement. Adapted from: [29]

Table 3.1.: Instructions required to achieve path as shown in Figure 3.5. Adapted from: [29]

Point	Instruction
p_0	MoveJ/MoveL
p_1	MoveC
p_2	
p_3	
p_4	MoveJ/MoveL

Online programming with the teach-pendant was the method applied in close to 90% of all robotic operations in 2019 [30]. One explanation might be that most technicians are familiar with the teach-pendant as most industrial robots are provided with this, making online programming a natural choice [28]. Further, programming with the teach-pendant is very precise and easy to use when programming simple trajectories. However, the main drawback with online programming is the stoppage in production while programming, which is an inevitable effect of online programming.

Offline programming does not use the teach-pendant nor any equipment near or inside the robot cell. This method simply requires a computer and a programming language. "Offline" implies that the actual programming can be executed at any time from anywhere, as opposed to online programming, which requires the physical robot. However, when talking about offline programming in robotics, one generally refers to advanced simulation software that allows for creating a digital three-dimensional representation of the robot cell. This digital representation of the robot cell is then used to program the robot's path. That being said, there is a difference between offline programming and simulation. Simulation does not imply offline programming. However, offline programming can not be done without simulation [31].

The digital and physical robot cells are closely related; either the digital cell is designed as an exact replica of the physical cell, or the physical cell is designed as an exact replica of the digital cell. Either way, the important thing is that they are exact replicas of each other. That being said, the digital cell should only include components that might interfere with the robot manipulator's trajectory. Components out of reach for the robot manipulator are not necessary to include as they will not impact the performance of the robot. However, they might impact the performance of the simulation.

There exist several methods for offline programming, the easiest being a so-called path generation [32]. This method is quite similar to online programming with the teach pendant. The programmed path is uploaded to the robot controller of the physical robot. As mentioned above, the digital and physical robot cells should be exact replicas of one another. The importance of keeping the discrepancies between the two to a minimum is related to the "quality" of the programmed path. One will have to make final adjustments after uploading the program to the physical robot [33]. However, the larger the deviation between the digital and physical cells, the more adjustments will have to be made after uploading program. This can take away part of the reason for choosing offline programming, namely the reduced production downtime.

One of the most important advantages of implementing offline programming is the increased efficiency in production by eliminating production downtime while

programming [28]. With online programming, the robot to be used for the final program is used for the programming as well, thereby causing production stoppage while being programmed. With OLP, production can continue while a new task is being programmed. According to Delfoi Robotics [34], production downtime caused by online programming can be reduced from up to a month, down to one day by replacing online programming with offline programming. This single day of production stoppage is related to the implementation of the program with the physical robot, as explained earlier in this section. This reduction in production downtime further entails a reduction in the cost associated with the downtime as well as the programming labour [31].

The reduction in programming stoppage accompanying OLP essentially means that high-mix, low volume production no longer is a constraint for robotic automation. With online programming, the time spent programming the robot could, in many cases, not be economically justified for companies with a rapidly changing production. Fortaco, an Estonia-based company that utilizes robots for welding operations, explains how online programming with the teach pendant is out of the question as the increased efficiency achieved by the robot would be lost in programming stoppage [35]. In opposed to online programming, offline programming is highly compatible with Lean methodology [34].

Further, OLP can assist in the decision-making when designing a robot cell for a specific need. It can be economically challenging for a company to invest in a robot that may or may not suit its needs. As mentioned earlier in this section, either the digital cell is designed to resemble the physical cell or vice versa. By developing the digital cell first, one can test the possible candidates with the simulation software prior to building the physical robot cell. This way, companies can find the optimal set-up for the robot cell digitally before installing it physically, and thereby avoid expansive and commonly made, mistakes by eliminating errors and flaws discovered through simulations [36]. Further, OLP contributes to a higher production rate as the operation is verified in advance [35]. As well as verification of the thought solution, OLP allows for testing different approaches to the same problem in order to find the best solution. This is not as easy with online programming, it is not justifiable to test many different approaches due to the time it takes to test only one [28].

Afrit, a trailer manufacturer, based in South Africa, explains how the implementation of offline programming has provided several benefits, such as improved repeatability, explained in Section 3.3, of welding as well as better consistency. Further, Afrit states that the time saved by replacing online programming with offline programming can be used to give attention to other aspects, such as improving welding [37].

Offline programming is less suited for simple operations with few points such as

pick and place, assembly, and packaging [31]. In such applications, it can be more cost-efficient to program online. However, for complex operations that require hundreds or even thousand points, online programming is not really an option; offline programming is necessary.

3.3. Simulation Software

The use of simulation software introduces many upsides to production. As mentioned in Section 3.2, the work cell can be designed in a simulation software before making the physical cell. This allows for analyzing the behaviour of the cell in advance of investing both time and money and for testing multiple scenarios and solutions. Common mistakes can be made in the simulation software and thereby avoided in the physical world [38]. Verification with regards to the robot´s reach and access can be done, as well as testing of end-effector tools, workpiece positioners, etc. Further, with simulation software, one can estimate cycle times and identify bottlenecks in the production line. Using simulation software is also advantageous with regards to the customer, as a visual presentation gives larger insight into the proposed solution. The simulation is also used to ensure that the final product will deliver as the end-user requires.

Even though the upsides are many, offline programming software has its challenges [31]. As mentioned in Section 3.2, regardless of the order in which they are designed, it is important that the digital and physical work cell are exact replicas of one another. This is related to the concept of a robot's repeatability and accuracy.

Repeatability, when talking about robotics, refers to the robot's ability to reach the same point over and over, whereas accuracy refers to the deviation between the desired point and the achieved point. The two concepts are illustrated in Figure 3.6. According to The Welding Institute, the repeatability of a typical industrial robot used in welding operations is ≤ 0.05 mm [39]. Further, a robot is able to follow its programmed path with an accuracy of 0.100 mm [40].

This implies that the robot will, with extreme precision, do exactly as programmed to. If, however, there is a position error of one or several components in the digital cell compared to the physical cell (or vice versa), the robot will not perform the task it was programmed to; it will perform the programmed path with a position error throughout. This motivates the introduction of robot calibration. The robot controller calculates the position of the end-effector based on a perfect kinematic model; however, the physical robot's kinematics are not perfect [41]. Robot calibration involves identifying the real geometrical parameters of the physical robots kinematics [42]. Offline programming can only reach its full potential if there is

a precise correspondence between the kinematics of the physical robot as well as the digital model. Several simulation software has tools for calibration.

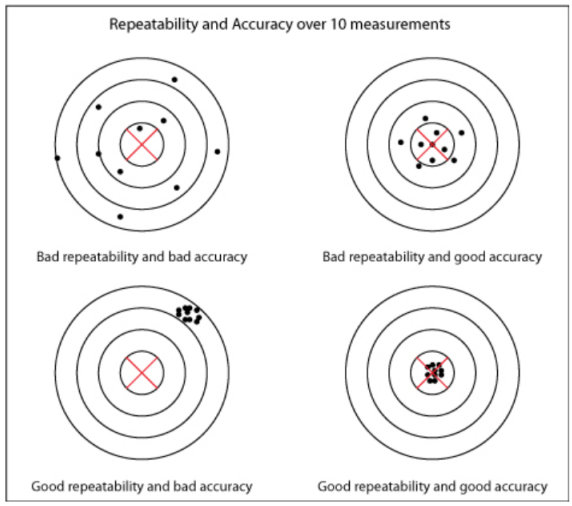


Figure 3.6.: Illustration of repeatability and accuracy plotted against each other [43]

When it comes to offline simulation software, the selection for the consumer is generous. Further, many of these simulation tools offer the same qualities, making it difficult to choose the right simulation software. However, one distinctive difference ultimately divides the entire selection into two groups; manufacturer-dependent and manufacturer-independent. Examples of simulation software from both categories are presented in the following sections.

3.3.1. Manufacturer-dependent software

Several robot manufacturers have developed simulation software compatible with their robots; ABB has their software called RobotStudio, KUKA has KUKA.Sim, and Yaskawa has their MotoSim. All robots and additional equipment produced by the manufacturer are usually provided in the component catalogue, making it easy to design the digital work cell. Such simulation software does, however, limit the user to that specific manufacturer. This can be problematic as companies often utilize robots from different manufactures in different operations, depending on their specific characteristics [35].

According to ABB, their simulation software, RobotStudio, is the world's most used offline programming software for robotics [44]. The software was designed to closely resemble the online programming method in order to ease the transition.

Their aim with the software is to reduce risks posed on both human and equipment with online programming, increase start-ups, decrease time spent on change-overs and ultimately achieve higher productivity. RobotStudio has advanced features, including virtual meetings, digital twins, explained in Section 3.1, Augmented Reality (AR) technology, Virtual Commissioning, and Stop Position Simulation. Referring to Section 3.1.3, ABB provides an OPC UA server compatible with their IRC5 robot controller.

KUKA's robotic offline programming software is called KUKA.Sim. They highlight features including planning reliability enabled by accurate cycle times, as well as collision detection and reachability checks [45]. The digital layout is easily designed by dragging the desired component into the three-dimensional workspace. Referring to Section 3.2, it was stated that offline programming is less suited for simple applications as, for example, pick and place. KUKA.Sim highlights both pick and place as well as automated palletizing as applications suitable with their simulation software. The software provides an OPC UA PLC interface that enables communication with several applications. Referring to Section 3.2, KUKA.sim has functionalities that enables implementation of Industry 4.0 [46]. Visual Components, an offline programming software to be presented in Section 3.3.2, was acquired by KUKA in 2017 [47]. Therefore, KUKA.Sim exhibits many of the same features as Visual Components, as well as the layout.

MotoSim EG-VRC is an offline programming software provided by Yaskawa. Their simulation software is capable of performing collision detection, reach analysis and cycle time calculations [48]. It supports process applications such as welding, cutting, and sealing. The offline programming process is identical to programming with the pendant, and programs can be downloaded directly to the compatible robot controllers. Three-dimensional CAD files of, for example, parts to be welded can be uploaded directly to the software without the need for conversion. Further, the robot path can be generated automatically based on information from the 3D CAD model.

3.3.2. Manufacturer-independent software

A problem with the brand-dependent offline programming software is that they limit the user to that brand. There does, however, exist different offline programming software that is independent of the robot manufacturer. Examples of such are RoboDK, Delfoi Robotics OLP, and Visual Components, which all offer an extensive library of robots from different manufactures. By implementing an offline programming software compatible with many different robot manufacturers, the company is able to use the same software for all operations. Fortaco explains how the use of one offline programming software in all operations is time-saving

Table 3.2.: Summary of key features with manufacturer-dependent simulation software [49], [50], [51], [52], [53], [54]

Comparison of manufacturer-dependent software				
Software	System requirements	Price	OPC UA compatible	Programming language
RobotStudio	Windows	US$1.500/year	Yes *	RAPID
KUKA.Sim	Windows	US$2.000	Yes	KRL
MotoSim	Windows	US$8.000	No **	INFORM

*IRC5 controller **MotoOPC software compatible with NX100, DX100 and FS100 controllers. No documentation for compatibility with OPC UA

as one only has to maintain one software, the operators only need to learn one software, and all operators have the same programming skills as they all use the same software [35].

RoboDK is an offline simulation tool for industrial robots, with an extensive library providing more than 500 robot manipulators from more than 40 different robots manufacturers [55]. Equipped with an intuitive interface, RoboDK states that no prior programming skills are necessary. Features include automatic optimizing of the robot path by avoiding singularities, axis limits and collisions. The software has calibration tools for improving accuracy, as mentioned at the beginning of this section. In order to export the final program to the robot at the shop floor, RoboDK is compatible with several robot controllers, such as ABB RAPID, KUKA KRC/IIWA, Motoman Inform, and Universal Robots, to name a few.

Delfoi Robotics provides software for robotic offline programming and simulation. According to them, they are one of the leading suppliers for industrial companies implementing offline programming in manufacturing [34]. As stated above, by implementing offline programming, high-mix, low volume production is no longer a constraint with robotic automation. According to Delfoi, robots in short-run production companies has increased from 30% to over 90%. Delfoi Robotics offline programming software supports robot manufacturers as Yaskawa, KUKA, ABB, Fanuc, and Staubli, to name a few. Their products include Delfoi ARC, Delfoi SPOT, Delfoi CUT, Delfoi PAINT, and Delfoi SURF-X [56]. The software focuses on easing the programming process, and, as with KUKA, Delfoi Robotics also uses software technology provided by Visual Components.

Visual Components is a Finland-based company founded by Scott Walter, Mika Anttila, and Juha Renfors in 1999 [57]. According to Scott Walter, their vision was to “make factory simulation software easy to use and affordable” [58]. Today, Visual Components is one of the leading companies within the industry of 3D manufacturing simulation software and is used, as stated above, by several major

actors in the simulation software industry.

Figure 3.7.: The stages of offline programming illustrated by Visual Components [59]

As with both Delfoi Robotics and RoboDK, Visual Components is a third-party software and offers more than 2500 components and over 60 robot brands. The software is used in many industries, such as the automotive-, electronic-, industrial automation-, and packaging and palletizing industry. With its extensive library of components, the software is easy to implement in diverse production lines that incorporates several brands. However, if a component is missing, or a specific work-piece is required, it is easy to import CAD models to the program, where Visual Components supports more than 20 CAD files [60]. In the "eCatalog", under Public Models, where all brands provided are listed, a model named Visual Components is available, containing components designed by Visual Components. This model essentially offers everything required to create a complete digital representation of a factory floor, ranging from different robots and advanced machines to walls and doors for enclosing the cell.

The software is divided into three categories; "Essential", "Professional", and "Premium" [61]. The Essential version allows for basic robot programming, virtual reality, and basic virtual commissioning to name a few. The professional version includes all features provided with the essential version, as well as the ability to convert CAD data into simulation models. The premium version includes all of the above-mentioned features and more advanced features within virtual reality and robot programming. Visual Components provide connectivity plugins for OPC UA, Siemens S7, SIMIT, Universal Robots RTDE, and WinMOD Net. WinMOD and SIMIT are only available for premium users. Further, the connectivity has been optimized with Premium 4.3, enabling better communication with other devices through, for example, OPC UA.

Table 3.3.: Summary of key features with manufacturer-independent simulation software [61], [62], [63], [64], [65]

Comparison of manufacturer-independent software				
Software	System requirements	Price	OPC UA compatible	Programming language
Delfoi Robotics	Windows	Not provided	Unknown	Unknown
RoboDK	Windows, Mac, Ubuntu	2995 Euros *	Yes	Python **
Visual Components	Windows	Not provided	Yes	Python ***

*Professional **RoboDK API ***Visual Components API

Chapter 4.

System Description of Robot Cell at NTNU

The robot at disposal for this project is the Yaskawa Motoman GP25-12. This chapter presents a system description of the robot cell installed at NTNU, as shown in Figure 4.1. This includes the robot manipulator, the welding equipment, and the enclosure. Further, the robot software allowing for offline programming is presented. Simple calculations on the robots forward- and inverse kinematics are presented in order to illustrate the concept explained in Chapter 2. This chapter has been gathered from project thesis [1], provided in the digital appendix, with some modifications throughout. Further, Section 4.1 has been extended, and Section 4.2 is new.

4.1. Industrial robot controller

It is important to distinguish between the controller and the teach pendant. The teach pendant provides easy access to the robot through a touchscreen display where one can both see and edit the available commands, obtain information about different joint variables, develop and edit programs, etc. The teach pendant, referred to as the programming pendant by Yaskawa [66], is equipped with a keyboard that allows for easy maneuvering in the form of, for example, "jogging", and online programming. Further, a button is installed at the back of the pendant, which must be pressed and held halfway in order to move the robot, and serves as a "live-man" switch by either being pressed- or released fully. This ensures safety while both testing and running programs. An additional emergency button is implemented in the upper left corner, important when the pendant is in remote mode, as it is with offline programming. Figure 4.2 shows the Motoman YRC1000 Industrial Robot Controller with the provided programming pendant in front of

Figure 4.1.: Yaskawa Motoman GP25-12 with welding equipment as installed at Perleporten

the controller.

The robot controller can be said to be the brain of the robot, seeing as the actual control lies within the controller. Code, also referred to as programs, developed either online through the teach pendant or offline via an external computer, are exported to the controller via a communication port such as, for example, an Ethernet connection. The exported program is translated to physical motions in the controller [67]. This information is then sent to the robots Central Processing Unit, CPU, which enables the robot to both process and run the program [68].

The YRC1000 controller has a non-windows based operating system and uses the language Inform III. It is equipped with standard communication interfaces, enabling connection with already exiting networks [69]. Two Ethernet ports are provided which, e.g. allows for connection to an external computer for offline programming. Further, the status of the controller can be read/set due to an FTP-capable TCP/IP web server function. The YRC1000 controller supports communication protocols such as OPC UA, explained in Section 3.1.3, enabling interoperability, explained in 3.1.

Figure 4.2.: Controller and programming pendant

4.2. Robot Software

The YRC1000 robot controller is compatible with many different programming methods, one of them being Robot Operating System, ROS [69]. This section will present a Python library written by supervisor Lars Tingelstad in order to communicate with the robot Yaskawa GP25-12 from an external computer, i.e. offline programming. The library translates information written in Python to ROS commands for the robot controller and essentially enables all the same functionalities as with the teach-pendant. As the library is quite extensive, this section has been limited to only present the functionalities explicitly used in this project. The repository is available at https://github.com/tingelst/moto/tree/main/moto. The necessary files must be installed on the robot controller, available at https://github.com/ros-industrial/motoman.

The "Moto" class contains the highest level API and is used, together with a class named "ControlGroupDefinition", to define the robot and connect to the robot controller with its respective IP address. This class requires a group id to be defined, which is a string describing the object. Further, a group number must be given, as the system is compatible with up to four control groups, e.g. several robots, and work-piece positioners, where 0 implies robot number one. Lastly, the number of joints must be defined, as well as their names.

The next class important to introduce is a class named "Motion". This class

provides methods required for the start-up process. A convenient but not vital method is “check_motion_ready”, which outputs whether or not the robot is ready for ROS commands. If, for example, the teach-pendant is not sat in remote-mode, explained in Section 3.2, which is required for OLP, the output message will print failure. Two important methods are “start_servos” and “start_trajectory_mode”. Both methods are self-explanatory; the former start the servos, and the latter starts trajectory mode, which is required prior to sending trajectory points to the controller.

When it comes to moving the robot, read joint positions from the robot, check the status of the robot, to name a few, a class called “Simple_message" is important. This class enables, amongst others, the above mentioned functionalities. In order to send trajectory points order to the robot controller, the class “JointTrajPtFull” must be presented. It takes seven inputs, the first being group number, explained above. Next, a sequence number must be defined, which states which number the current point is in the total trajectory. The third, “Valid_fields” is always set to “ValidFields.TIME | ValidFields.POSITION | ValidFields.VELOCITY”. A time must be given, which states the time at which the robot should be at that position. The last three variables; position, velocity, and acceleration, all require input as a list of integers, and the list must contain ten elements as the software is compatible with robots that have up to ten degrees of freedom. As the Yaskawa GP25-12 has six joints, the first six elements correspond to the six joints. The last four are set to zero.

In order to send a trajectory point to the robot controller, the method “send_joint_trajectory_point(trajectorypoint)” must be entered, where the input “trajectorypoint” is the point one desires to send. Lastly, to retrieve information about the position, velocity or acceleration of the robot, a class “Joint_feedback” is provided.

4.3. The robot manipulator

The Yaskawa Motoman GP25-12 is a robot manipulator with six rotational joints connected by seven links, as can be seen in Figure 4.1. Referring to Chapter 2, this robot has six DOF. This implies that the GP25-12 has a configuration space and a task space of dimension six. As the dimension of the configuration space is equal to the dimension of the task space, this robot is not kinematically redundant. However, the robot will have configurations in which singularities arise [11]. Yaskawa Motoman labels each axis as S, L, U, R, B, and T, which stands for Swing or Swivel, Lower arm, Upper arm, Rotate, Bend, and Twist, respectively [70]. Six axes allow for the robot manipulator to exhibit the movement of a human arm. Each axis has limitations with regard to maximum motion range and maximum

speed. The repeatability, discussed in Section 3.3, of this robot is ± 0.03 mm. Further, the maximum work range is 2010 mm. The above-explained specifications as well as other important variables are summarized in Table A.2 and Table A.3 in Appendix A.2.

4.3.1. Kinematic calculations

A simple program calculating the forward- and inverse kinematics for the Yaskawa Motoman GP25-12 has been developed in order to illustrate the concepts explained in Section 2. The code editor Visual Studio Code, VS-code, was used with the programming language Python. The packages "numpy" and "modern_robotics" were installed using pip install.

The zero-position for the GP25-12 is as shown in Figure A.1 in Appendix A.2, where the respective lengths from joint one to joint six can be identified as; l_1 = 505 mm, l_2 = 760 mm, l_3 = 200 mm, l_4 = 150 mm, l_5 = 275 mm, l_6 = 807 mm, and l_7 = 100. Note that this includes an offset between joint one and two, resulting in seven identified lengths. A simplified sketch of the robot has been developed in order to illustrate the concept, shown in 4.3.

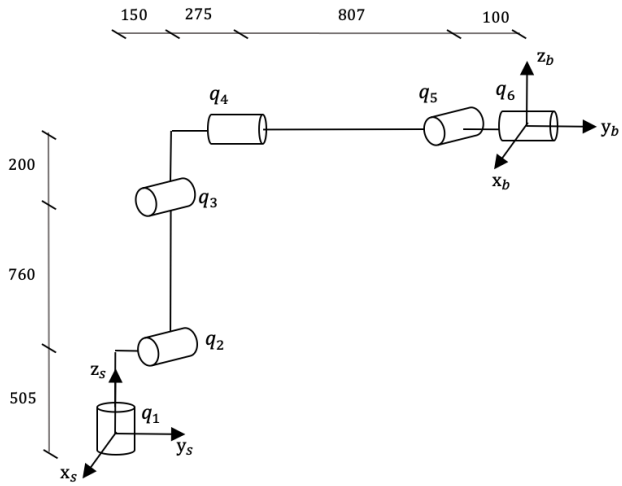


Figure 4.3.: Simplified sketch of Yaskawa GP25-12 in its zero-position

The global coordinate system is defined using the right-hand rule with positive x axis pointing out of the paper, positive z axis pointing upwards, resulting in positive y axis pointing in the direction of the spherical wrist. The M matrix representing the end-effector configuration when the robot is at its zero-position,

explained in Section 2.4.2, is found to be

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & l_5 + l_5 + l_6 + l_7 \\ 0 & 0 & 1 & l_1 + l_2 + l_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1332 \\ 0 & 0 & 1 & 1465 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.1}$$

with lengths l_1 - l_6 as above.

However, with the welding gun mounted on the wrist, the new end-effector position will be at the tip of the welding gun. To obtain this configuration, the homogeneous transformation matrix described in Section 2.1 was used. Let {b} denote the frame at the wrist, and {t} denote the frame at the tip of the welding gun. By studying Figure 4.4 one can see that frame {t} relative to frame {b} is rotated some amount about the x_b axis and translated some amount in y- and z direction. These values were obtained from the programming pendant. The transformation matrix expressing the configuration of the new end-effector was found to be

$$M_{\text{tool}} = T_{\text{bt}} = \begin{bmatrix} R_{\text{bt}} & p \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.59 & 0.81 & 450.27 \\ 0 & -0.81 & 0.59 & 84.40 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.2}$$

where $R_{\text{bt}} = \text{Rot}(\mathrm{x}, \theta) = \text{Rot}(\mathrm{x}, -0.94)$ and $p = [\mathrm{x}, \mathrm{y}, \mathrm{z}] = [0, 450.27, 84.40]$. Note that the angle given in radians is negative, due to the previously defined coordinate frame $\{\mathrm{x}, \mathrm{y}, \mathrm{z}\}$ explained earlier in this section.

The screw axes represented in space form, $\mathcal{S}_i = (\omega_i, v_i)$, and in body form, $\mathcal{B}_i = (\omega_i, v_i)$, can be found as explained in Section 2.4.2. The vectors ω_i and v_i are presented in Table 4.1 in space form and Table 4.2 in body form. Each screw axis is the 6×1 column vector formed by the respective ω_i and v_i. The screw axes were calculated in VS-code as shown in Listing A.1 line 41 to 71, and 81 to 112 in Appendix A.3.

To compute the forward kinematics solution, two functions in the "modern_robotics" library can be utilized; FKinSpace and FKinBody. Both functions take as input the matrix M as well as a list of joint angles, named "thetalist" by default. The list of angles will be a 1×6 vector for the GP25-12 as it has six joints. The last input depends on whether the screw axes were calculated in space form or

Figure 4.4.: Close-up of wrist and the attached welding gun

Table 4.1.: Vectors ω_i and v_i for i=1 to 6 for Yaskawa GP25-12 represented in space form

i	ω_i	v_i
1	(0,0,1)	(0,0,0)
2	(1,0,0)	(0, l_1, -l_4)
3	(1,0,0)	(0, l_1+l_2, -l_4)
4	(0,1,0)	(-(l_1+l_2+l_3),0,0)
5	(1,0,0)	(0, l_1+l_2+l_3, -(l_4+l_5+l_6))
6	(0,1,0)	(-(l_1+l_2+l_3), 0, 0)

Table 4.2.: Vectors ω_i and v_i for i=1 to 6 for Yaskawa GP25-12 represented in body form

i	ω_i	v_i
1	(0,0,1)	(-(l_7+l_6+l_5+l_4),0,0)
2	(1,0,0)	(0, -(l_3+l_2), l_7+l_6+l_5)
3	(1,0,0)	(0, -l_3, l_7+l_6+l_5)
4	(0,1,0)	(0,0,0)
5	(1,0,0)	(0, 0, l_7)
6	(0,1,0)	(0, 0, 0)

body form. The function FKinSpace takes a list of screw axes calculated in space form, named “Slist” by default. The function FKinBody takes a list of screw axes calculated in body form, named “Blist” by default. These lists can be calculated in VS-code as shown in Listing A.1 in Appendix A.3, line 71 and 112, respectively.

The resulting Slist and Blist for the GP25-12 are shown in Equation (4.3) and (4.4), respectively.

$$\text{Slist} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1465 & 0 & -1465 \\ 0 & 505 & 1265 & 0 & 1465 & 0 \\ 0 & -150 & -150 & 0 & -1232 & 0 \end{bmatrix} \tag{4.3}$$

$$\text{Blist} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ -1332 & 0 & 0 & 0 & 0 & 0 \\ 0 & -960 & -200 & 0 & 0 & 0 \\ 0 & 1182 & 1182 & 0 & 100 & 0 \end{bmatrix} \tag{4.4}$$

For inverse kinematics calculations, the library "modern_robotics" uses the Newton-Raphson method explained in Section 2.5.2. As with the forward kinematics, there exist two functions for computing the inverse kinematics; IKinSpace and IKinBody. Both functions take in the matrix M, a matrix describing the desired end-effector configuration, a list of guessed joint angles, and an error allowance "eomg" for orientation and "ev" for position. Again, the last input depends on whether the screw axes were calculated in space form or body form. IKinSpace takes Slist as input, and IKinBody takes Blist as input, both lists being equal to the ones derived above.

To calculate the inverse kinematics for the GP25-12, the transformation matrix given as the solution to the forward kinematics was set as the desired end-effector configuration, T = T_goal. Then, an initial guess of the joint angles was stored in a list called "thetalist_guess". The error allowance for orientation was set to 0.001, and the error allowance for position was set to 0.0001.

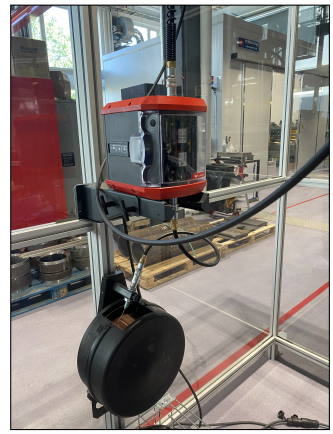
The two functions will output a list of angles required to obtain the desired end-effector configurations and a result of the iterations, which will be either "true" or "false". The result would be true if the iterations were able to find a solution within the defined error allowances and false if not. If the initial guess of angles is too far from the solution, the iterations will not converge, and the result will be false.

4.4. Welding equipment

A Fronius CMT welding system is installed with the Yaskawa robot manipulator. CMT is a welding technique that utilizes very low heat input and as a result shields the material to be welded from property changes, amongst others. The CMT welding system from Fronius works by detecting a short circuit. During welding execution, the short current is supervised by the digital process control, and at occurrence, the welding wire is pulled back. This short circuit control results in low temperatures [71].

More specifically, the welding equipment from Fronius installed at the lab is the MIG/MAG power source TPS 400i as can be seen in Figure 4.5a. The wire feeder is shown in Figure 4.5b. The TSP 400i is equipped with a touch screen, which provides the operator with system information, and the operator is also able to perform adjustments such as welding parameters. The technical data for the TPS 400i is provided in Table A.1 in Appendix A.1.

(a) Fronius TPS 400i



(b) Wire feeder

Figure 4.5.: Image a) displays the Fronius TSP 400i and image b) the wire feeding system

The welding gun is attached to the robot´s wrist joint as an additional piece of equipment, where the tip of the welding gun makes out the end-effector of the robot manipulator. The wiring from both the wire feeder and the power source is gathered on the robot prior to the spherical wrist. The wires provided to the welding gun are implemented within the spherical wrist, allowing for a more flexible movement.

4.5. Welding cell enclosure

A welding process exposes the operator and those in near proximity to a hazardous environment. Security measures as welding helmets shielding the operator, and curtains shielding those around must be implemented. However, automating the process by introducing robots adds a new safety hazard; a moving robot.

Different solutions for enclosing the welding cell exist. The Yaskawa Motoman GP25-12 with Fronius TPS 400i is enclosed by see-through walls, as shown in Figure 4.6. A lock mechanism is installed at the door, which is implemented in the software of the robot. This functions as a safety measure where the operator will not be able to execute programs in remote mode unless the door is closed. This protects the operator, and anyone near the robot cell, from both dangers related to the welding operation, such as spatter and damaging light, and the robot manipulator itself.

Figure 4.6.: Welding cell with enclosure

Chapter 5.

Development of the suggested solutions

This chapter will present the development and implementation of offline programming with the robot manipulator Yaskawa GP25-12. The required installations will be described, followed by a presentation of the virtual robot cell designed based on the physical robot cell at Perleporten. The rest of the chapter is divided into two main parts; the development of the OPC UA server and the development of a solution for sending the offline programmed trajectory to the physical robot.

This project consists of four main components; a computer, an offline programming simulation software, the OPC UA protocol, and the robot cell at Perleporten. A presentation of different offline simulation software was given in Section 3.3. However, the software Visual Components was chosen for this project before startup.

The computer hosts the simulation software and serves as the link between the digital robot cell in Visual Components and the physical robot cell. The OPC UA protocol ensures communication between the computer and Visual Components. Referring to Section 3.1.3, the computer contains the OPC UA server, whereas Visual Components serves as the OPC UA client.

The communication between the physical robot and the computer is achieved by an Ethernet connection between the computer and the robot controller. The set-up is shown in Figure 5.1.

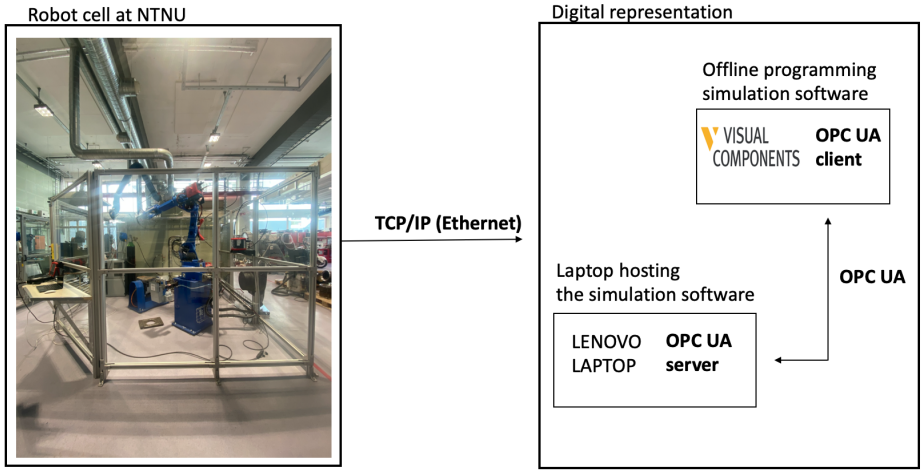


Figure 5.1.: Components in project

5.1. Installations

Visual Components Premium 4.3 was installed and activated with a license provided by NTNU. Further, VS-code was chosen as the software in which code would be written. This coding editor is compatible with several coding languages, including Python, used in this project. With this coding editor, a "pip install" command can be used to install necessary packages.

The Github repository published by the OPC Foundation, explained in Section 3.1.3, does not provide any resources compatible with Python. An open-source OPC UA library, available at Github `https://github.com/FreeOpcUa`, provides an extensive amount of resources compatible with both Python and C++, and offers a wide range of examples. The repository “python-opcua” was used as a basis for this project.

In order to use the functionalities provided in the free OPC UA open-source Github library, they must be installed as “pip install opcua”. The same yields for the software translating Python code to ROS commands for the robot controller, explained in Section 4.2. The installation was executed as “pip3 install git+https://github.com/tingelst/moto.git –upgrade”.

To connect to the robot controller, an Ubuntu terminal was installed on the computer hosting the simulation software.

5.2. Virtual robot cell

As stated in Section 3.2, there are two ways of creating a digital work cell in a simulation software; either the digital cell is made based on an already exiting physical cell, or the digital cell is made prior to the physical cell. For this project, the physical cell was already installed. The digital and physical robot cell are presented in Figure 5.2 and Figure 5.3, respectively.

As mentioned in Section 3.2, components that do not interfere with the robot manipulator are not necessary to include in the digital representation, as these might impact the performance of the simulation. As can be seen in Figure 5.3, the robot manipulator is placed on a positioner, elevating the robot manipulator from the floor. This component was included in the digital representation, as it affects the performance of the digital representation with regards to the physical cell. The eCatalog in Visual Components did not have this exact component. However, as its only function is to elevate the robot, a similar component with the same height was included in the digital cell.

The workpiece positioner in the physical cell was also not provided in Visual Component´s eCatalog. Yaskawa was contacted in an attempt to obtain CAD files for the components missing in Visual Components. As it turned out, this would be a rather time-consuming process, and seeing as the workpiece positioner would not affect the progress of the project, it was decided to proceed without it. A similar positioner was included in the digital representation for illustration.

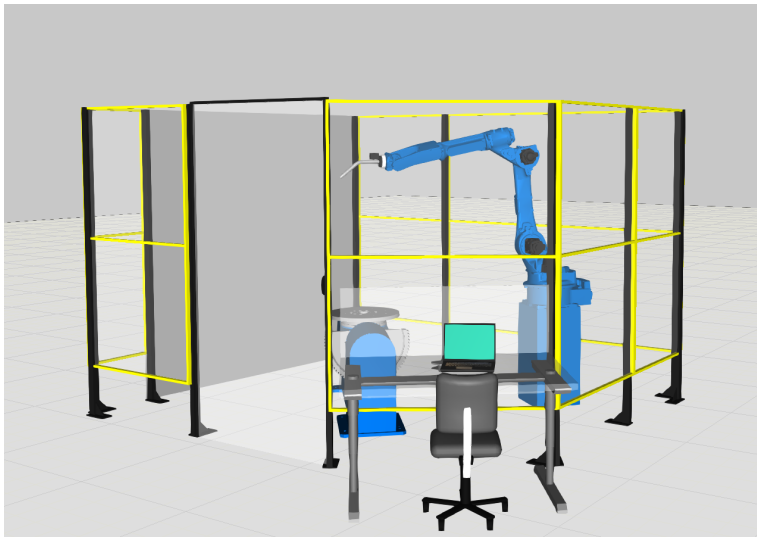


Figure 5.2.: Digital representation of robot cell

Figure 5.3.: Physical robot cell

5.3. Testing the OPC UA communication

As explained in Section 3.3.2, Visual Components has connectivity features as add-on, which have to be enabled in the configurations. With the OPC UA connectivity enabled and the software restarted, the OPC UA server was available in Visual Components. In order to test the connection, the example server "server_minimal.py" provided by the Free OPC UA library was used. In the server, one must define the server endpoint, as explained in Section 3.1.3, with an IP address and a port number. As both the server and client were located on the same computer, the IP address was set to "localhost" with port number 4840. This is a commonly used port number in OPC UA servers; however, not a required one. For the connectivity configurations in Visual Components, an OPC UA server address must be defined. The server address was set to be equal to the string stored in the server endpoint in the server. The server endpoint in the OPC UA server and the server address in Visual Components must be identical.

The OPC UA server in Visual Components consists of two parts; "Server to

simulation" and "Simulation to server". Both of these contains variables from the digital environment as well as variables from a connected server. The server "server_minimal.py" creates an object called "MyObject" and adds a writable variable called "MyVariable" to this object. The server increments a variable "count" every 0.1 seconds and sets "MyVariable" to this value.

With the server running in VS-code, the connection in Visual Components changed to True, implying that a connection was established between the server and client. With the connection established, the object "MyObject" was available in the "Simulation to server" configuration. The variable "MyVariable" within "MyObject" from the server was paired with the variable "S" from the simulator, representing joint S of the Yaskawa GP25-12. The simulation in Visual Components was started, resulting in the simulated robot rotating about joint S, verifying that the communication was successful.

With the OPC UA connection between the server and Visual Components established, the next step was to modify the server to have six writable variables corresponding to the robot's six joints. The server "server_minimal.py" generates a value that is stored in the variable "MyVariable". Besides being useful for testing purposes, this generated value serves no purpose. A more interesting value to store in these writable variables would be the actual joint positions, read from the physical robot.

A new server was created, based on "server_minimal.py", the master´s thesis written by Aksel Øvern [72] , and the Moto library developed by supervisor Lars Tingelstad. The code can be seen in Listing B.1 in Appendix B.

The new server creates an object called "Moto", similar to "MyObject", explained above. Further, all six joint links are defined as writable variables which are added to the "Moto" object.

As explained in Section 4.2, some essential methods defined in the Moto library must be implemented in order to achieve movement with the physical robot. In the server script, above the initiation of the server, the physical robot was defined. Further, the servo and trajectory mode was started. The joint angles of the physical robot were extracted using the command "robot.state.joint_feedback(0).pos". Here, "robot" is the object containing the physical robot, explained above.

In order to retrieve information from the physical robot, a connection between the computer and the robot controller must be established. Referring to Section 5.1, an Ubuntu terminal was installed to connect with the robot controller through an Ethernet cable. In the Ubuntu terminal, the following must be entered: "telnet 192.168.255.200". If a connection is established, Ubuntu requests a username and password. Once the connection with the robot controller was established and the server initiated, Ubuntu outputs that the robot is ready for ROS commands, see

Figure 5.4.

```
thea@NTNU08363: ~
Starting new connection to the Motion Server
Creating new task: IncMoveTask
IncMoveTask Started
Creating new task: tidAddToIncQueue (groupNo = 0)
Creating new task: tidAddToIncQueue (groupNo = 1)
Creating new task: tidMotionConnections (connectionIndex = 0)
Starting new connection to the State Server
Starting new connection to the IO Server
Creating new task: tidIoConnections (connectionIndex = 0)
Setting servo power: 1
In StartTrajMode
Robot job is ready for ROS commands.
Robot job is ready for ROS commands.
```

Figure 5.4.: Successful connection to robot controller through Ubuntu. Servos and trajectory mode started

Following the same procedure as before, a connection was established between the server and Visual components. The six joint variables from the server were paired with their respective joint variables in the simulation, see Figure 5.5. Once all joint variables were paired correctly, the simulation was initiated in Visual Components, and the simulated robot moved to the desired position, see Figure 5.6.

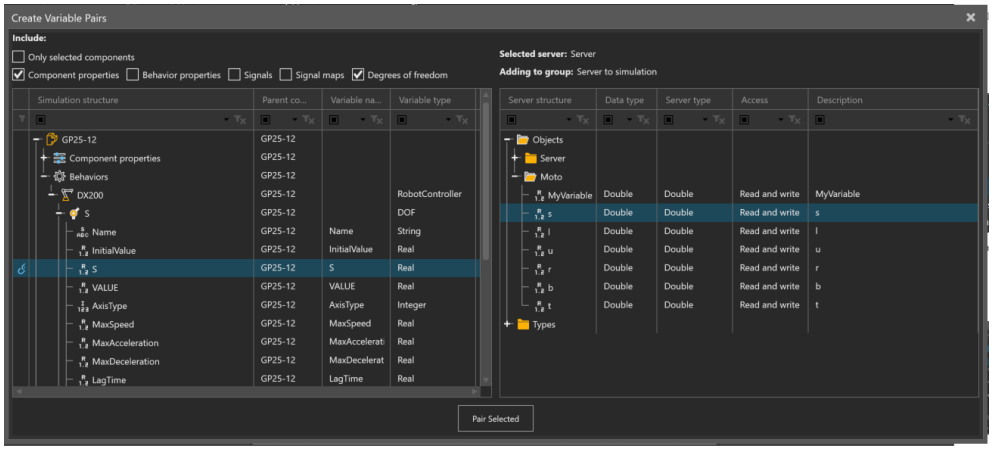


Figure 5.5.: Pairing joint variable S from server to simulation in Visual Components

Important to note from this test was that Visual Components takes joint angles in degrees, whereas the physical robot requires joint angles in radians, implying that it also output joint angles in radians. The position of the robot in Visual Components, shown in Figure 5.6, actually displays degrees. The values are correct, but the robot barely moved. Converting the joint angles read from the physical robot to degrees before sending these to Visual Components made the simulated robot move to the desired position.

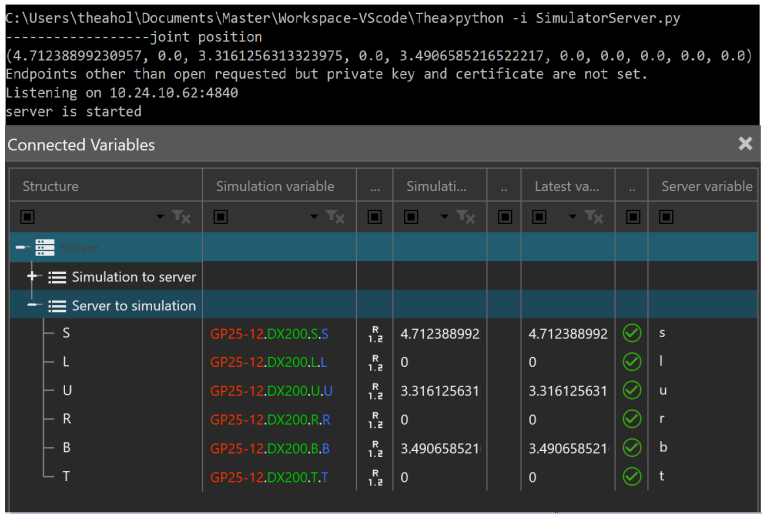


Figure 5.6.: Top image displays the joint positions read from the physical robot and sent to Visual Components, and the bottom image displays the joint positions of the simulated robot corresponding to the ones of the physical robot

5.4. Offline programming from simulation to server

The OPC UA connection was proven to be successful, and the simulated robot mirrored the position of the physical robot. The next step was to reverse the process, to get the physical robot to mirror the position of the simulated robot, i.e. offline programming. The process of implementing offline programming with the Yaskawa GP25-12 required an extensive amount of testing. Essential was to understand how best to utilize the functionally provided with the Moto library, as well as perfecting the OPC UA server, reading joint positions from Visual Components. The following sections will present the process of developing Python scripts, testing, and improvements.

In order to send a trajectory to the robot controller, the Moto library explained in Section 4.2 had to be investigated thoroughly. A server had to be created that connects to the client, reads joint positions, and sends these to the physical robot. Reading joint positions from Visual Components was easily achieved by changing "set_value" to "get_value" in the server. However, sending these joint positions to the robot proved to be challenging.

Two methods for implementation with the robot controller were tested. The robot software implemented in this project is not compatible with "real-time" programming. Therefore, the initial idea was to write the joint positions from client to file, before sending these to the physical robot. The other idea was to

write directly to the robot controller from Visual Components, however, with some delay.

5.4.1. Developing the OPC UA server

In Visual Components, a simple trajectory consisting of four points was programmed. The initial point recorded was with the robot manipulator in its zero-position, explained in Section 4.3.1. This implies that all joint angles are equal to zero. Next, three more points were placed at arbitrary positions. All points were programmed with a "point-to-point Motion Statement" allowing the robot to move all joints freely. Visual Components automatically generated the path from point p1 to point p4.

The joint variables were paired as before, only now from simulation to server. In order to store the positions read from Visual Components, a JSON file was created. The server gets joint angles from Visual Components every 0.2 seconds and creates a 1×6 position vector that stores these joint angles before writing the position vector to file.

In the server, all variables in the object "Moto", explained in Section 5.3, are given a default value of 6.7. As soon as the server starts, if there are no constraints on the writing-to-file functionality, 1×6 position vectors with all six elements equal to 6.7 will be written to file until the simulation in Visual Components is initiated. At this moment, the joint angles read from Visual Components will be written to file. However, as the position vectors containing the default value 6.7 are not part of the programmed trajectory, they can not be included in the file. A test was executed in order to illustrate the problem described above. The result is shown in Figure 6.1 in Section 6.1.

To account for this, an IF-statement was included in the server, which is only entered if the value of one of the joint variables is different from 6.7, implying that the simulation in Visual Components is initiated.

Further, the server will get values from the client as long as the server is running. This implies that even though the simulated trajectory is finished, the server will continue writing position vectors to file. A solution to this problem was to create a method in the server that checks whether or not the simulated trajectory is completed, see Listing C.3, line 23 to 43, in Appendix C.3.

The above-described modifications to the server were crucial in order to obtain an accurate joint position file containing the offline programmed path. Two more, however less important, constraints were implemented in the server. An IF statement was included in order to stop writing to file when the simulation in Visual

Components is resat, as well as a statement that clears the content of the JSON file between each test.

With the suggested server, a problem regarding the relation between the simulation speed, and the speed at which the server gets values, was discovered. The simulation speed in Visual Components can be adjusted from 0.0 to 10000. If the speed of simulation is high, the server will have to get values at a very high rate in order to obtain enough position vectors along the trajectory. If the speed at which the server gets values is not sufficiently high, a lot of points along the path will be lost from simulation to server. This results in a trajectory that deviates from the programmed one, ultimately making it useless.

When programming a path in Visual Components, one can assign a so-called "Cycletime" to each point along the trajectory. This can be compared to the variable "time", explained in Section 4.2, which resembles the time given the physical robot to reach the desired position. By giving such a cycle time to each point in the simulation, the server read-time can be adapted to this and thereby controlling the process. With a cycle time of 5 seconds and a server read-time of 0.5 seconds, a total of ten position vectors will be read by the server as the simulated robot moves from point A to B, given that the simulation speed is 1.0. This implies that every tenth position vector written to file corresponds to a point in the simulation, see Figure 5.7. The nine previous ones record the movement from the previous point to the point in question, ensuring that the physical robot will move exactly as the simulated robot also in between each programmed trajectory point.

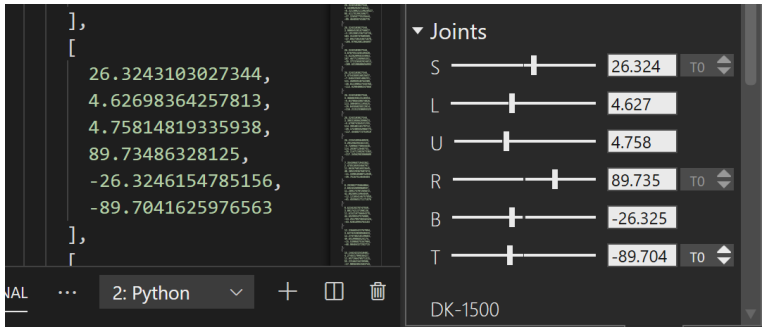


Figure 5.7.: Tenth position vector written to file (left) corresponding to the joint angles of the first trajectory point in Visual Components (right)

5.4.2. Developing script for implementing OLP with robot controller

A separate script was developed in order to send the offline programmed trajectory to the robot controller. As explained in Section 4.2, the robot controller requires one to send a trajectory point, p0, recording the robot´s current position before sending a trajectory point, p1, instructing a new position. At the early stages, a misinterpretation concerning sending trajectory points to the robot controller was made. It was understood that point p0, recording the current position of the robot, had to be sent to the robot controller before a new trajectory point, p1, every time. This was later realized to be a misconception and is commented on later in this section.

The file that stores the offline programmed trajectory consist of one large vector containing all the position vectors read from simulation. A for-loop was created in order to retrieve the position vectors one by one and store these in a variable. The position vectors from simulation are 1×6. However, as explained in Section 4.2, when defining a trajectory point, the position-, velocity-, and acceleration vectors must be 1×10. An array containing four zeros were therefore added to the end of each position vector from file and stored in a new variable to be given as input to the trajectory points.

As explained in Section 5.4.1, the first point in the trajectory programmed in Visual Components was with the robot in its zero-position, i.e. all joint angles equal to zero. This was done as a security measure for when the trajectory would be implemented with the physical robot. Based on the amount of uncertainties related to the implementation of offline programming with the robot controller in question, it was decided that it would be advantageous to always know the first position vector sent to the robot controller. This allows for the physical robot to, prior to receiving positions from file, be moved to the zero position from any arbitrary position under controlled conditions.

To ensure that the physical robot was in the zero position before receiving the offline programmed trajectory points, a simple “check” was included that calculates the difference between the robot´s current position and the zero position, essentially a 1×10 zero vector. If the difference is smaller than a defined threshold, this is interpreted as the robot being in the zero position, and a point p0, recording the current position of the robot, is sent to the robot controller. A difference larger than the threshold implies that the current position is too far from the zero position, and the robot must be moved. Based on the current interpretation of how to send trajectory points to the robot controller, three points were included; a point p0 that records the current position of the robot, followed by a point p1 that moves the robot to zero position over a given time period. Lastly, a point p0

was sent to the robot controller again, updating the current position of the robot.

With the robot in the zero position, the offline programmed trajectory could be sent. Two points were created within the for-loop presented above; a point, p0, recording the current position of the robot, followed by a point, p1, that takes the 1×10 position vectors explained above as input. The server is shown in Listing C.1 in Appendix C.1. Running this script made the robot move; however, lots of errors from the robot controller were displayed in Ubuntu. These errors stated that the trajectory start position did not match the current position. As stated earlier in this section, the functionality of the robot software was misinterpreted. After discussing the results and errors with supervisor Lars Tingelstad, it was realized that the point recording the current position, p0, only has to be sent one time as the first point of the trajectory. Based on this new insight, the script was modified.

The IF-statement entered if the robot was too far from zero position was modified to only contain two points; point p0 reading current position and point p1 moving the robot to zero position. Further, the point p0 within the for-loop was removed, as this would already have been sent to the robot in the IF-statement above, leaving only the trajectory point, p1, reading position vectors from file. When testing the modified version, the physical robot moved a small amount before stopping while displaying "excessive segment" on the teach-pendant. It was realized that the sequence number, explained in Section 4.2, of the trajectory points produced in the for-loop was set to 1 throughout the for-loop. This means that while the position vector in the trajectory point changed, the sequence number remained at 1, essentially telling the robot that the first point in its trajectory has infinite many positions, which is impossible. A variable called "sequence_nb" was created that starts at zero and is incremented by one for each for-loop.

Another important realization regarding the functionality of the trajectory point was made while testing the script. The "time" variable, explained in Section 4.2, was initially interpreted as a time given the robot to perform the specified movement; if "time" was set to 0.5 seconds, each generated trajectory point in the for-loop should take 0.5 seconds. However, the time variable functions similarly to the "sequence" variable described above. It is not simply a time given to execute the current point; it describes at what time the robot should be at the desired position. As with sequence = 1, if time = 0.5 in all trajectory points generated in the for-loop, this tells the robot that it should be at all these different positions at the exact same time, which is impossible. A variable "timer" was created, which is incremented for every for-loop. This variable is not, however, necessarily initiated at zero. If the robot is not in the zero position and the IF-statement, explained earlier, is entered, the robot is given 5 seconds to move to the zero position. This means that the variable "timer" in the for-loop must start at 5,

and be incremented a desired amount. If the robot is in zero position, meaning the IF-statement is not entered, the variable “timer” starts at zero. The resulting script is shown in Listing C.4 in Appendix C.4.

5.5. Testing of the finalized versions

As explained in Section 5.4.1, the main challenge with the developed solutions was to find the optimal relationship between the speed at which the server gets values and the speed of the simulation in Visual Components. Further, an additional factor regarding speed was realized. In the script sending the offline programmed trajectory to the robot controller, there needs to be some sleep in the script in order for the physical robot to "keep up". Finding the optimal setup required testing. As stated at the beginning of this chapter, two different implementation methods were tested; writing from simulation to file before sending to the robot controller and writing directly to the robot controller from simulation.

5.5.1. Sending position vectors directly to robot controller

A script was developed that connects to the robot controller and Visual Components at the same time, see Listing C.2 in Appendix C.2. This script incorporates the OPC UA server explained in Section 5.4.1, and the functionalities for sending trajectory points to the robot controller, explained in Section 5.4.2. However, instead of writing the offline programmed trajectory to file, it was temporally stored in a vector and sent directly to the robot controller.

In Visual Components, two identical robots were placed next to one another, see Figure 5.8. One of the robots were programmed to execute a simple trajectory, and the variables were paired from simulation to server. The other robot was paired from server to simulation. Both pairings are shown in Figure 5.9. Within a for-loop, the server gets joint position values from the first robot, store these in a temporary vector, and immediately send them to the robot controller. Simultaneously, the current position of the physical robot is retrieved and written to simulation. The desired outcome was for the physical robot to, with some delay, mirror the movement of the first simulated robot, while the second simulated robot received position vectors from the physical robot and simulated the actual movement of the physical robot.

This test was not successful. Sending the offline programmed trajectory directly from simulation to the robot controller worked well; however, reading the position vectors from the physical robot and writing these to simulation did not. A recording was made that shows the physical robot mirroring the movement of the simulated robot, with some delay. The result is shown in Section 6.2.

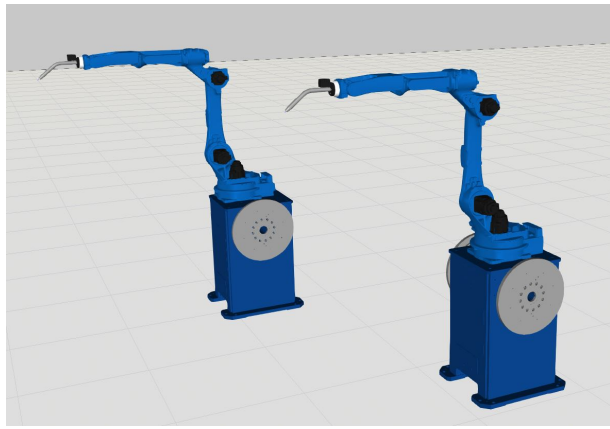


Figure 5.8.: Simplified digital cell for testing simulation to server and server to simulation simultaneously

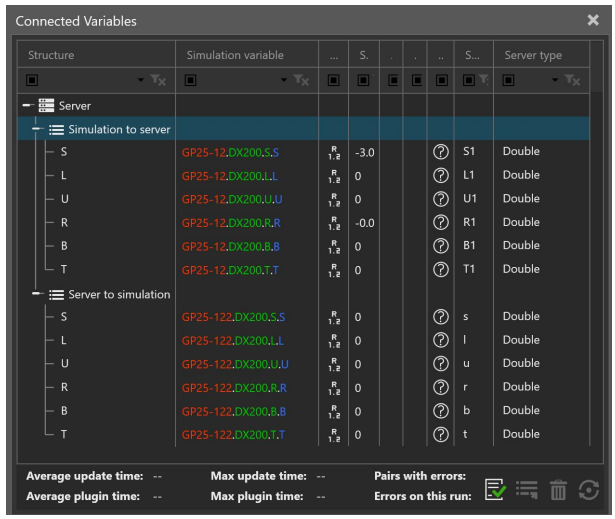


Figure 5.9.: Paired variables from simulation to server and from server to simulation

5.5.2. Sending position vectors to robot controller from file

The second implementation method tested was developed with two separate scripts. An OPC UA server, as explained in Section 5.4.1, and a script for sending trajectory points to the robot controller, explained in Section 5.4.2. From the tests executed with the above-explained implementation method, it was realized that the communication with Visual Components and the communication with the robot controller have different “requirements” proven difficult to satisfy at the

same time. The two scripts are shown in Listings C.3 and C.4, in Appendix C.3 and C.4, respectively. The desired outcome was to find the optimal relationship between the server read-time and the sleep in the script sending trajectory points to the robot controller.

The first tests were executed with the sleep after the robot has moved to zero position set to 2 seconds, while the "time" variable in the trajectory point instructing the robot to move to zero position was set to 5 seconds. The tests in Figure 6.2, 6.3, and 6.4 were executed with sleep after each trajectory point set to 0.3, 0.1, and 0.2, respectively.

The next tests were executed with the sleep after the robot has moved to zero position equal to the "time" variable given to arrive at desired position; 5 seconds. The tests in Figure 6.5, 6.6, and 6.7 were executed with sleep after each trajectory point sat to 0.2, 0.3, and 0.4, respectively.

Finally, a test was executed to check the repeatability of the offline programmed trajectory. The same trajectory was sent to the robot five consecutive times, and for each test, the position of the robot was read and written to file. Figure 6.8 shows all five tests in the same plot, while Figure 6.9 shows the same figure, only zoomed in on the areas where discrepancies between the five tests were at a maximum. Table 6.1 displays the largest difference encountered for each joint in all five tests.

Chapter 6.

Results

This chapter will present the results obtained from testing the developed solutions. The results will be described and commented on in Chapter 7. The results are presented in the same order as the solutions were presented in Chapter 5.

All plots display six sub-figures, one for each of the robot´s six joints. The unit "step" along the x-axis represents the total amount of position vectors in the trajectory, implying that "step 0" represents the first point in the trajectory. The unit "angle" along the y-axis is given in radians.

6.1. Results from testing OPC UA server without constraints

This section presents the result obtained when comparing two trajectory files; the desired trajectory programmed in Visual Components and the actual trajectory written to file from the OPC UA server. The result is shown in Figure 6.1.

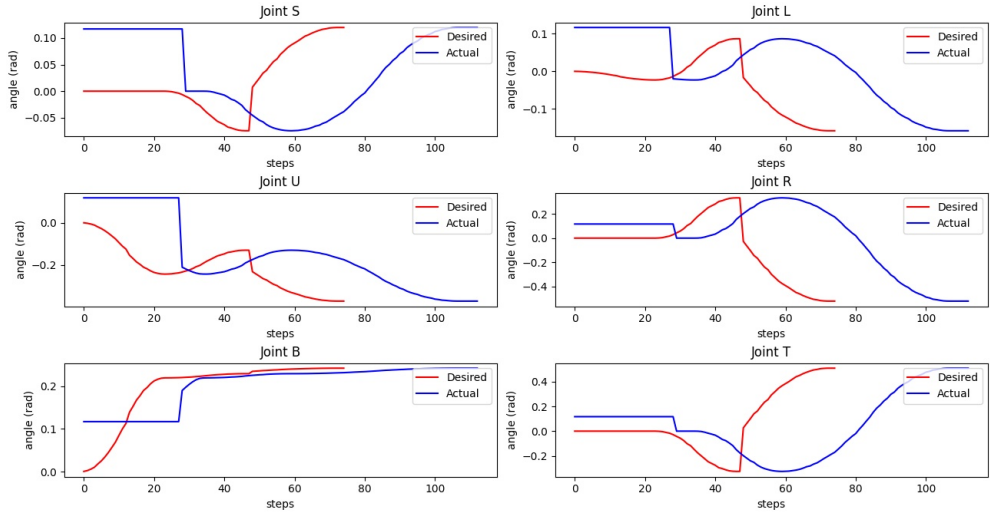


Figure 6.1.: The red line illustrates the programmed trajectory in Visual Components, while the blue line illustrates the trajectory written to file without constraints in the server

6.2. Results from sending trajectory directly from Visual Components to the robot controller

A video is provided in the digital appendix, illustrating the result when sending trajectory points directly from Visual Components to the robot controller. The recording displays the physical robot mirroring the movement of the simulated robot, with some delay.

6.3. Results from sending trajectory to file and then to robot controller

The results presented in this section were obtained with different configurations with regards to sleep in the script. The robot was moved to its zero position prior to receiving trajectory points from file in all tests. The time given to arrive at zero position was 5 seconds.

Figure 6.2, 6.3, and 6.4 displays the results obtained with sleep after moving the robot to its zero position equal to 2 seconds. Figure 6.5, 6.6, and 6.7 displays the results obtained with sleep after moving the robot to its zero position equal to the time given to move to zero position; 5 seconds. This information is provided in the figure description as “time” = 2 for the first three, and “time” = 5 for the

three last.

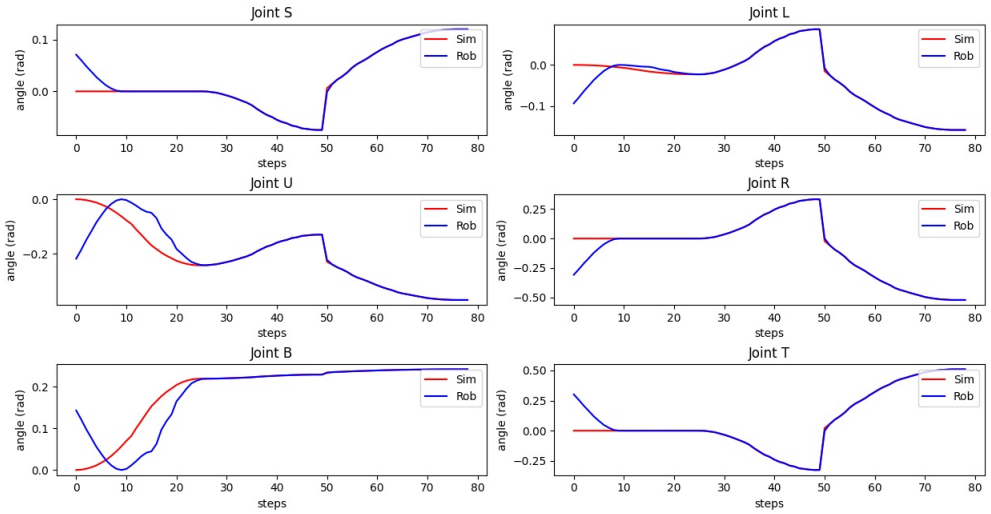


Figure 6.2.: Result obtained with "time" = 2, and sleep after each trajectory point sent to the robot equal to 0.3

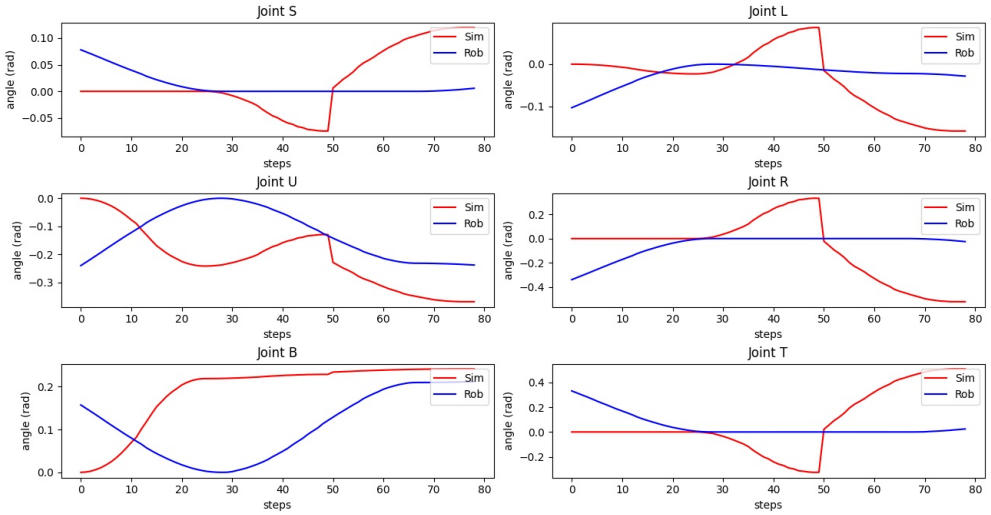


Figure 6.3.: Result obtained with "time" = 2, and sleep after each trajectory point sent to the robot equal to 0.1

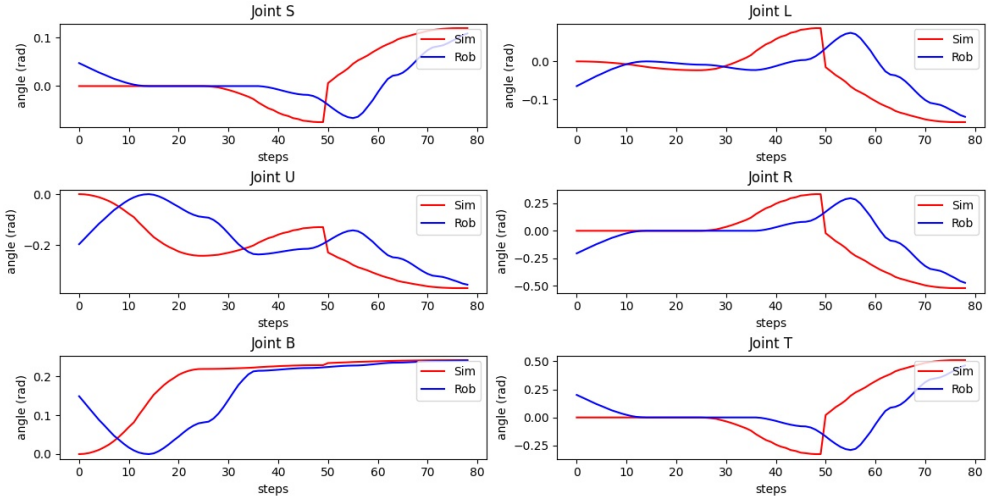


Figure 6.4.: Result obtained with "time" = 2, and sleep after each trajectory point sent to the robot equal to 0.2

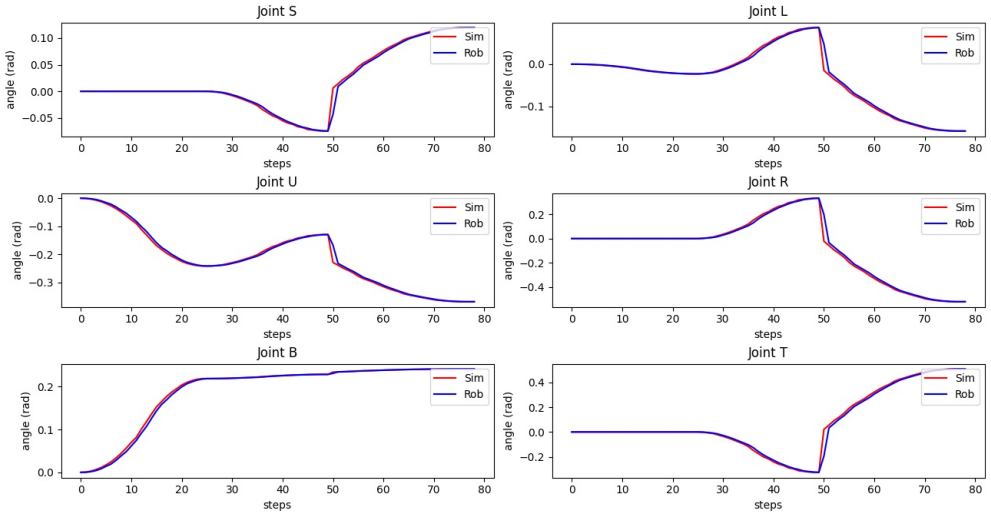


Figure 6.5.: Result obtained with "time" = 5, and sleep after each trajectory point sent to the robot equal to 0.2

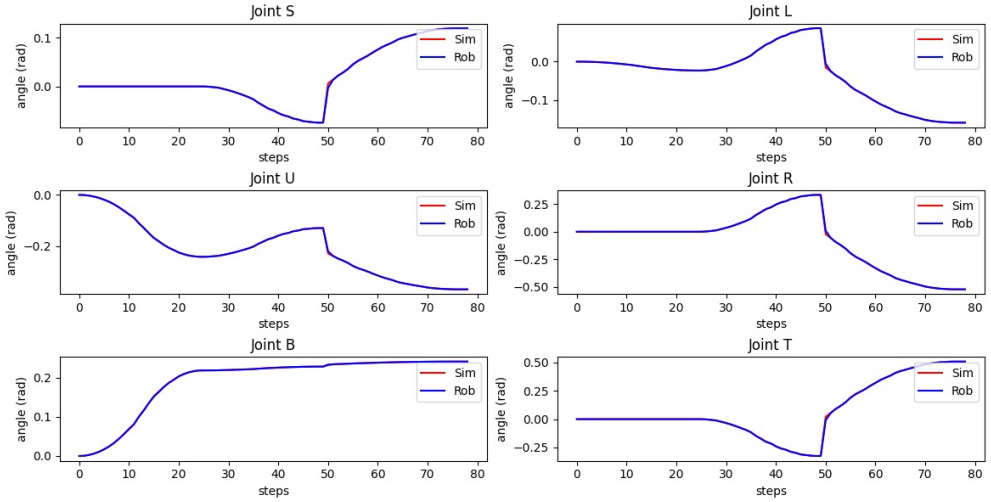


Figure 6.6.: Result obtained with "time" = 5, and sleep after each trajectory point sent to the robot equal to 0.3

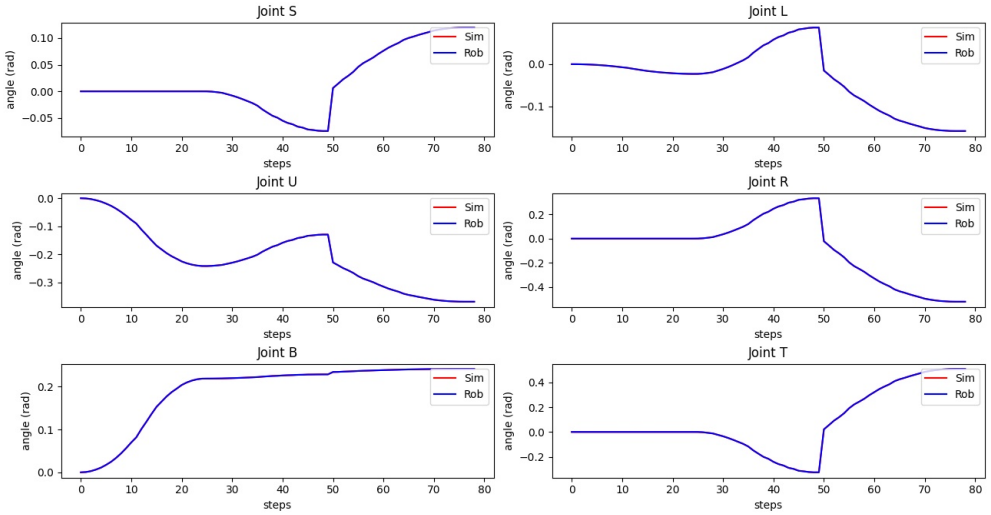


Figure 6.7.: Result obtained with "time" = 5, and sleep after each trajectory point sent to the robot equal to 0.4

6.4. Checking the accuracy

This section presents the results obtained when sending the same trajectory to the robot controller five consecutive times. Figure 6.8 displays all five tests in the same plot. Figure 6.9 is the same plot as displayed in Figure 6.8, only zoomed in on areas with the largest error between the five tests. The labels 1, 2, 3, 4, and 5 represent test 1 to 5, respectively.

Table 6.1 displays the largest error found between the minimal joint angle and the maximal joint angle in all five tests, see Figure 6.9.

Table 6.2 and 6.3 presents the minimal and maximal error between the trajectory sent to the robot controller and joint feedback received from the robot controller in Figure 6.5 and 6.7, respectively.

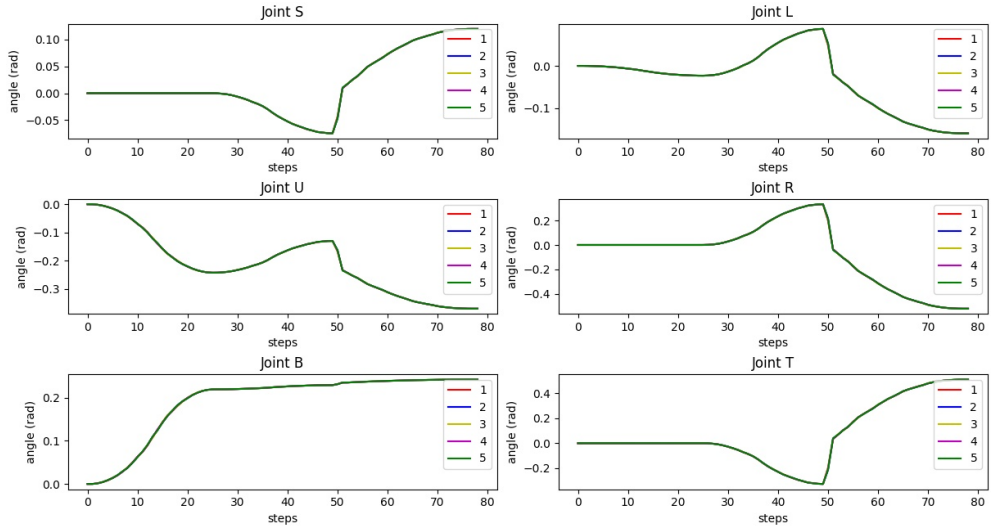


Figure 6.8.: Result obtained when sending the same trajectory file to the robot controller five consecutive times. Labels 1-5 represent test 1-5, respectively

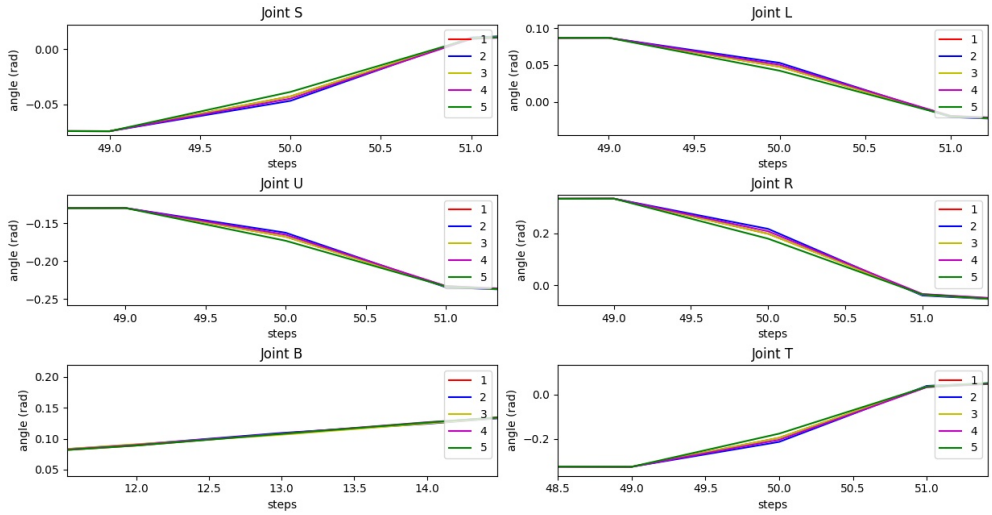


Figure 6.9.: Figure 6.8 zoomed in on areas with small discrepancies between the five consecutive tests. Labels 1-5 represent test 1-5, respectively

Table 6.1.: Largest difference recorded for each joint in five consecutive tests. All values are in radians

Joint	Step	Min [rad]	Test	Max [rad]	Test	Difference [rad]
S	50	-0.046932284	Test2	- 0.038852174	Test5	0.008080110
L	50	0.042341530	Test5	0.052772656	Test2	0.010631125
U	50	-0.172967285	Test5	0.162360907	Test2	0.010906378
R	50	0.179402381	Test5	0.217299730	Test2	0.037897348
B	13	0.106403485	Test3	0.109252304	Test2	0.002848819
T	50	-0.213544220	Test2	-0.176354364	Test5	0.037189856

Table 6.2.: Comparison of the simulated trajectory and the feedback from physical robot in Figure 6.5. The values are in radians and given in absolute value

Joint	Min error [rad]	Step	Max error [rad]	Step
S	0.0	–	0.049204747	Step51
L	0.0	–	0.062942668	Step51
U	0.0	–	0.061745344	Step51
R	0.000014887	Step73	0.220588276	Step51
B	0.000005147	Step74	0.011258153	Step114
T	0.0	–	0.216544803	Step51

Table 6.3.: Comparison of the simulated trajectory and the feedback from physical robot in Figure 6.7. The values are in radians and given in absolute value

Joint	Min error [rad]	Step	Max error [rad]	Step
S	0.0	–	0.000047506	Step51
L	0.000000091	Step70	0.000034207	Step51
U	0.0	–	0.000068365	Step51
R	0.0	–	0.000271847	Step28
B	0.000002075	Step31	0.000363389	Step51
T	0.0	–	0.000539283	Step27

Chapter 7.

Discussion

Even though to some, Industry 4.0 may merely be a buzzword, there is no arguing that an impressive advancement in the manufacturing industry is happening. Technology is evolving at a rapid speed, and the information flow is greater than ever. Industry 4.0 essentially enables the demand for high-mix, low volume production, as the exchange of information is constant, implying that adaption to new information is possible at all time.

Robots have been implemented successfully in mass-production for a long time. However, with the new demands introduced with Industry 4.0, they are on the verge of becoming outdated. The method of programming robots online is not compatible/sustainable with an ever-changing request from the consumer. If the time required to program the robot with the teach pendant equals the time it takes for the demand to change, the robot will never reach the stage of production. This might be a rather blunt statement, but as explained by Delfoi Robotics in Section 3.2, online programming can take everything from two weeks to a month. It is not unlikely that demands change from one month to the next.

Robotic offline programming offers one major advantage being that the production stoppage is drastically decreased, as the programming is executed from an external computer not dependent on the physical robot. Not only does this increase efficiency, it essentially revitalize robots in Industry 4.0. While the robot is producing the exciting demand, the new demand can be programmed, enabling the optimal production strategy.

The fact that offline programming is not dependent on the physical robot while programming is highly relevant with the current pandemic. The ability to work from home while maintaining progress has never been more important. While the current pandemic won´t last forever, it is interesting to consider the effects it will have on the industry. As home-office has become the new normal, it is reasonable to assume that companies see a potential in economical savings by not having

their employees in office every day of the week.

Further, the pandemic introduced a drastic increase in demand for products normally not desired by the "average" person, such as sanitary products, e.g. face masks and antibacterial gel. Companies that were able to adapt to these changes quickly undoubtedly profited from it, while others who did not might want to change their strategy in order to be more versatile to new demands. Based on the literature and the factors discussed above, a fair assumption is that robotic offline programming will increase in the years to come.

The literature describes robotic offline programming, almost exclusively, by all the advantages it introduces. However, referring to Section 3.2, close to 90% of all robotic operations in 2019 were programmed online. Possible reasons for this might be related to, what some would call, misconceptions regarding the method. Whether or not they are, is a topic of discussion.

One reason why some companies might be opposed to implementing offline programming can be related to the belief that it requires expertise. In this project, referring to Section 3.2, the digital robot cell was designed to replicate the already exciting physical robot cell. As explained in Section 5.2, several components installed in the physical cell were not provided in the eCatalog in Visual Components. With Visual Components, one can create and upload 3D models of components designed in, for example, SolidWorks. However, this requires the operator to have, or to obtain, competence with CAD design.

Further, most simulation software advertise themselves as being easy to use, where no prior programming skills are required. Understanding the basics of Visual Components is manageable. However, based on personal experience, the documentation in general is not very user-friendly. The simulation software is extremely advanced, which is both positive and negative. Positive because it really does enhance the quality of a production cell both in the design phase and the production phase, with the insight and advanced programming functionalities it provides. However, in order to achieve the full advantage of the software, at least for someone new to simulation software, it is a plausible assumption that quite some time must be invested.

The OPC UA protocol used in this project is advanced. It consists of very specific technology that most people are unfamiliar with, including people in the IT industry. Referring to the discussion above, implementing such a technology requires knowledge within the subject in order to fully take advantage of its features. Even though the OPC UA protocol is not a requirement with offline programming, considering the statements above, this motivates the question; is robotic offline programming becoming so advanced, compared to online programming, that it essentially requires a completely new set of skills?

That being said, implementing a simple OPC UA server with Visual Components was easy, and the connection worked seamlessly. As both the OPC UA server and the entire Moto library were written in Python, no problem arose regarding the compatibility of all three software components. Issues that were faced with the developed solutions were not directly caused by one of the software components but rather how to best incorporate them with one another.

As explained in Section 5.4.1, constraints on the read-functionality of the server had to be implemented. Figure 6.1 in Section 6.1 displays the results without constraints on the server. The programmed trajectory sent to file is shown in red, and the actual trajectory written to file in blue. It can be seen that all six joints maintain a constant joint angle for about 30 steps; this is the default value given in the server. One step represents one position vector, implying that close to 30 position vectors were written to file before the simulation in Visual Components was initiated. This further explains the delay in the plots, as the trajectory from simulation is added after the 30 position vectors created with the default value in the server.

As soon as the simulation is initiated, all joint angles should change to the angle of the desired trajectory at step 0. This is the case for joint S, L, R, and T. However, joint U and B are far from the respective angles in step 0 of the desired trajectory. This can be explained by the quite drastic change in joint angle from step 0 to around step 20, which only occurs in joint U and B. The remaining four joints have quite a small, if any, change in angle from step 0 to 20. The server read time is too slow compared to the speed of the simulation, resulting in the rapid change in joint angles being lost from simulation to server. The same issue can be seen with the joint angle change in joint S, L, U, R, and T at around step 50. In the desired trajectory, these joint angles change during a few steps, whereas the actual trajectory requires around 20 steps to achieve the same angle.

The trajectory from file has around 110 steps, whereas the actual trajectory has about 75. The actual path should have 30 steps more than the desired one due to the default values written to file in the beginning. However, the actual trajectory has around 5 steps more than it should. This is because the server gets values from the simulation even though the trajectory is finished and only stops when the server connection is manually closed.

Two methods of implementation were tested with the final solutions. The first one being to send position vectors directly to the robot controller from Visual Components, explained in Section 5.5.1. This implementation method was successful with regards to sending the offline programmed trajectory to the robot controller. The result is provided as a digital recording in the digital appendix. However, retrieving joint position feedback from the robot was unsuccessful. As the communication with the robot in this test was achieved while the server was running,

it was highly affected by the speed of the server. The problem with this method can be explained by the script essentially running too fast compared to the movement of the robot. The trajectory points are sent to the robot controller every 0.2 seconds and temporarily stored while awaiting their execution time. However, the joint feedback from the robot controller is retrieved equally fast. This implies that the joint feedback from the robot ultimately reads the same position over and over.

It can be argued that the unsuccessful result from this test is not of great importance; it is important with regards to logging the movement of the physical robot, and it is essential in order to simulate the movement in Visual Components. However, it is not essential, nor does it affect the offline programmed trajectory.

The second method of implementation tested was writing the trajectory to file before sending it to the robot controller, explained in Section 5.5.2. Referring to Figure 6.2, 6.3, and 6.4 in Section 6.3, in all three tests the joint angles of the robot, shown in blue, have a position error in step 0, compared to the simulation. This can be explained by the lack of sleep in the script.

The robot was given 5 seconds to move to its zero position prior to receiving trajectory points from simulation, while a sleep of 2 seconds was given after sending this trajectory point. The problem is that the trajectory point is sent to the robot controller, and then the script pauses for 2 seconds. The robot, however, will still use 5 seconds to perform the movement. This entails that when the simulated robot is at step 0 with all joint angles equal to zero, the physical robot is still 3 seconds away from its zero position.

Figure 6.3 shows the worst result of the three tests. However, the actual movement of the robot was smooth. The time to execute each trajectory point was set to 0.2 seconds while the sleep after each trajectory point was set to 0.1 seconds. The same problem as previously explained arises; the script runs faster than the robot is moving. For every position vector sent to the robot, the joint feedback from the robot is received when the robot has executed 50% of the trajectory point.

Figure 6.4 shows the result when the sleep after each trajectory point is equal to the time given in each trajectory point. This test best illustrates the problem arising when the robot is not at the correct start position before receiving trajectory points; there will be a lag throughout the entire trajectory.

Three tests were executed with sleep after moving the robot to zero position equal to the time given to move to zero position; 5 seconds. The results are shown in Figure 6.5, 6.6, and 6.7. Figure 6.5 illustrates an impressive improvement from Figure 6.4, where the only modification made is that the script waits while the robot moves to zero position before continuing. Table 6.2 displays the minimum and maximum error obtained between the simulated and actual trajectory. The

largest error for all joints, except joint B, was found in step 51, where a rapid change in angle takes place. When analyzing the plots, there seems to be a trade-off between the accuracy and the smoothness of movement.

Figure 6.7 displays the absolute best correspondence between the simulated trajectory and executed trajectory. Referring to Table 6.3, the largest errors are all very small. However, the robot did not execute the trajectory in one smooth movement but rather in sequences of movement. If the robot were to be implemented in a welding operation it would definitely be problematic for the result of the weld if the robot moves in segments rather than one smooth movement. The same is true for Figure 6.6. The common factor with these two tests was that the sleep after each trajectory point was set to be higher than the time given to execute the movement. The smoothest execution while still performing accurately was found with sleep after each trajectory point equal to the speed of execution, see Figure 6.5.

One possible source of error is important to discuss. The quality of executed trajectory, as seen in the plots presented above, range from highly inaccurate to highly accurate. However, these plots are based on the received joint feedback from the robot controller, and as discussed, this is highly dependent on the speed of the script. When the sleep after a trajectory point is larger than the time given to execute the movement, the received joint feedback is very accurate. When the sleep after each trajectory point is less than, or equal to, the time given to execute the movement, the plots displayed less accurate results. However, this does not necessarily imply that the robot does not move as desired, but rather that the suggested solution is not completely trustworthy with regards to analyzing the joint feedback.

Referring to Figure 6.9 and Table 6.1 in Section 6.4, the largest deviations between five consecutive tests for all joint angles, except for joint B, is found to be in step 50. By studying Figure 6.8, one can see that all joints, except joint B, experience a drastic change in joint angle around step 50. The concept of repeatability was presented in Section 3.3. As the five consecutive tests show good results, where the largest difference between two tests was found to be 0.038 radians, it is reasonable to assume that the robot´s repeatability explains these discrepancies seen in Figure 6.9.

Further, from Table 6.1, it can be seen that the smallest joint value and the highest joint value, i.e. the largest deviation, for all joints in step 50 regards test 2 and test 5, except the smallest value in joint B. Although it is not clear why these two tests are furthest apart in all joints in step 50, it can be argued that it is preferable for the results to be consistent rather than random. This consistency in joint deviations between tests indicates that the robot moves very accurately within one trajectory.

Chapter 8.

Conclusion

This report has presented the process of achieving the main goal for this project; to implement offline programming with the Yaskawa Motoman GP25-12 installed at Perleporten, NTNU. Further, four objectives were defined in Section 1.1.

Referring to the first objective, aspects of the industry supporting the implementation of robotic offline programming was presented in 3, and assessed in 7. With the evolvement in digital connectivity accompanying Industry 4.0, there is unquestionably a need for robotic offline programming. The OPC UA protocol solves the issue of interoperability, and the amount of simulation software available motivates the conclusion that offline programming is implementable for all. However, the fact is that robotic offline programming has not replaced online programming.

In Chapter 7, it was discussed why this might be, where the topic of complexity dominated, ultimately leading to the question: "is robotic offline programming becoming so advanced, compared to online programming, that it essentially requires a whole new set of skills?". Based on the three software components; the Moto library, the OPC UA protocol, and Visual Components, used to develop a solution for implementing offline programming with the Yaskawa GP25-12, my answer would be yes.

A digital representation of the physical robot cell was designed in Visual Components Premium 4.3, according to objective number two in Section 1.1. Some components were missing in the eCatalog in Visual Components. However, the lack of these did not influence the project.

The main focus during the project was within the three software components and how to best incorporate these. Referring to objective three, a solution was to be developed that incorporated Visual Components, OPC UA, and the Moto library. Both methods of implementation presented in this report incorporate all three components; however, the first of the two is especially interesting with regards to

the compatibility. With only one script, communication was achieved with Visual Components and the physical robot simultaneously, where the robot mirrored the movement of the simulated robot, with some delay.

However, with the suggested solutions, it is concluded that the second method of implementation presented is the best of the two. Taken into consideration is the importance of receiving joint feedback, which is not possible with the first method presented.

Joint feedback from the physical robot was given much attention in this project. It was essential in order to evaluate the performance of the suggested solution, referring to the fourth and last objective. From the best result obtained, the largest position error found in one joint was 0.0005 radians. The smallest error found was 0.0. Further, between five consecutive tests, the largest difference found between the angles at one given point in the trajectory was 0.038 radians, whereas the smallest difference was 0.003 radians. These numbers substantiate the conclusion that the implementation of offline programming with the Yaskawa GP25-12 was successful.

8.1. Further Work

One issue that needs to be solved is the issue of speed. The first method that sends trajectory points directly from simulation implements all its functionalities within the OPC UA server, implying that all information read from Visual Components and sent to the robot controller is controlled by the speed of the server. A form of "await" functionality could be implemented to slow down the script in order to receive joint feedback from the physical robot.

Further, as explained in Section 5.4.2, the physical robot was moved to its zero position prior to receiving trajectory points, and the first trajectory point programmed in Visual Components was always with the simulated robot in zero position. This solution is not ideal. An attempt to avoid the issue could be to read the position of the physical robot, write this position to Visual Components and implement this position as the first point in the offline programmed trajectory.

References

[1] T. Holmedal, "The automated welding industry; how it has been achieved and the challenges it faces," Project Thesis, Dec. 2020.

[2] K. M. Lynch and F. C. Park, *Modern Robotics: Mechanics, Planning and Control*. Cambridge University press, 2019.

[3] T. Rowland, *Orthogonal matrix*. [Online]. Available: https://mathworld.wolfram.com/OrthogonalMatrix.html.

[4] Motion Controls Robotics, *Unraveling degrees of freedom and robot axis: What does it mean to have a multiple axis pick and place or multiple axis robot?* [Online]. Available: https://motioncontrolsrobotics.com/unraveling-degrees-of-freedom-and-robot-axis-what-does-it-mean-to-have-a-multiple-axis-pick-and-place-or-multiple-axis-robot/.

[5] J. Iqbal, M. Ul Islam, and H. Khan, "Modeling and analysis of a 6 dof robotic arm manipulator," *Canadian Journal on Electrical and Electronics Engineering*, vol. 3, pp. 300–306, 2012. DOI: https://www.researchgate.net/publication/280643085_Modeling_and_analysis_of_a_6_DOF_robotic_arm_manipulator.

[6] S. Kucuk and Z. Bingul, "Industrial robotics: Theory, modelling and control," in. Germany/ARS, Austria: Pro Literatur Verlag, 200§, vol. December, ch. 4, pp. 117–147.

[7] P. Corke, *Paths and trajectories*. [Online]. Available: https://robotacademy.net.au/masterclass/paths-and-trajectories/.

[8] T. Kunz and M. Stilman, "Turning paths into trajectories using parabolic blends," Georgia Institute of Technology, Tech. Rep., 2011.

[9] Robotic Industries Association, *Defining the industrial robot industry and all it entails*. [Online]. Available: https://www.robotics.org/robotics/industrial-robot-industry-and-all-it-entails.

[10] N. Correll, *Advanced robotics 4: Inverse kinematics*, Feb. 2012. [Online]. Available: http://correll.cs.colorado.edu/?p=1958.

[11] Mecademic, *What are singularities in a six-axis robot arm?* [Online]. Available: https://www.mecademic.com/resources/Singularities/Robot-singularities.

[12] P. Corke, *Inverse kinematics and robot motion.* [Online]. Available: https://robotacademy.net.au.

[13] A. Fox, *Why you know more about industry 4.0 than you think*, Dec. 2019. [Online]. Available: https://www.nist.gov/blogs/manufacturing-innovation-blog/why-you-know-more-about-industry-40-you-think.

[14] TIBCO Software Inc., *What is industry 4.0?* [Online]. Available: https://www.tibco.com/reference-center/what-is-industry-4-0.

[15] Spectral Engines GmbH, *Industry 4.0 and how smart sensors make the difference*, Feb. 2018. [Online]. Available: https://www.spectralengines.com/articles/industry-4-0-and-how-smart-sensors-make-the-difference.

[16] B. Marr, *What is industry 4.0? here´s a super easy explanation for anyone*, Sep. 2018. [Online]. Available: https://www.forbes.com/sites/bernardmarr/2018/09/02/what-is-industry-4-0-heres-a-super-easy-explanation-for-anyone/.

[17] Kalycito Infotech Pvt Ltd, *What is opc ua, industry 4.0 and the interoperability challenge.* [Online]. Available: https://www.kalycito.com/opc-ua-interoperability/.

[18] OPC Foundation, *History.* [Online]. Available: https://opcfoundation.org/about/opc-foundation/history/.

[19] OPCconnect.com, *History of opc.* [Online]. Available: https://www.opcconnect.com/history.php.

[20] Novotek, *Opc and opc ua explained.* [Online]. Available: https://www.novotek.com/uk/solutions/kepware-communication-platform/opc-and-opc-ua-explained/.

[21] A. Frejborg, M. Ojala, L. Haapanen, O. Palonen, and J. Aro, *Opc ua connects your systems - top 10 reasons why to choose opc ua over opc*, May 2013. [Online]. Available: https://downloads.prosysopc.com/downloads/automation_xx_seminar_opcua_connects_your_systems.pdf.

[22] OPC Foundation, *Unified architecture.* [Online]. Available: https://opcfoundation.org/about/opc-technologies/opc-ua/.

[23] Kalycito Infotech Pvt Ltd., *What is opc ua?* [Online]. Available: https://www.kalycito.com/opcua/#fourth.

[24] Unified Automation GmbH, *Opc ua nodeid concepts*. [Online]. Available: https://documentation.unified-automation.com/uasdkhp/1.4.1/html/_l2_ua_node_ids.html.

[25] Exor International, *The introduction of opc ua publish-subscribe and its importance to manufacturers*, Sep. 2019. [Online]. Available: https://www.exorint.com/en/blog/the-introduction-of-opc-ua-pubsub-publish-subscribe-and-its-importance-to-manufacturers.

[26] Real Time Automation, Inc., *Opc ua overview*. [Online]. Available: https://www.rtautomation.com/technologies/opcua/.

[27] OPC Foundation, *Opc foundation announces opc ua open source availability*. [Online]. Available: https://opcfoundation.org/news/opc-foundation-news/opc-foundation-announces-opc-ua-open-source-availability/.

[28] A. Owen-Hill, *What are the different programming methods for robots?* Mar. 2016. [Online]. Available: https://blog.robotiq.com/what-are-the-different-programming-methods-for-robots.

[29] Yaskawa America, Inc, *Dx200 operator's manual for spot welding using motor gun*, 2015. [Online]. Available: http://assets.efc.gwu.edu/yaskawa-motoman/165297-1CD.pdf.

[30] S. Jeannet, *5 ways to program a robot*, Jun. 2019. [Online]. Available: https://www.motoman.com/en-us/about/blog/5-ways-to-program-a-robot.

[31] T. M. Anandan, *Demystifying robot offline programming*, Sep. 2018. [Online]. Available: https://www.automate.org/industry-insights/demystifying-robot-offline-programming.

[32] PPMA ltd, *Robot programming methods*. [Online]. Available: https://www.ppma.co.uk/bara/expert-advice/robots/robot-programming-methods.html.

[33] M. Castor, *The need for robotic offline programming in a covid-19 world*, Dec. 2020. [Online]. Available: https://www.controleng.com/articles/the-need-for-robotic-offline-programming-in-a-covid-19-world/.

[34] Delfoi Robotics, *Offline programming robot simulation and offline programming*. [Online]. Available: https://www.delfoi.com/delfoi-robotics/offline-programming/.

[35] ——, *Fortaco - efficient robot welding for high mix- low volume production*, Oct. 2020. [Online]. Available: https://www.delfoi.com/references/robotics-references/fortaco-efficient-robot-welding-for-high-mix-low-volume-production/.

[36] Blumenbecker Group, *Offline programming*. [Online]. Available: https://www.blumenbecker.com/industrial-automation/industrial-robotics/offline-programming.

[37] Delfoi Robotics, *Afrit production now runs faster than ever*, Jun. 2020. [Online]. Available: https://www.delfoi.com/afrit-production-now-runs-faster-than-ever/.

[38] B. Brumson, *Robotic simulation and off-line programming: From academia to industry*, Nov. 2009. [Online]. Available: https://www.automate.org/industry-insights/robotic-simulation-and-off-line-programming-from-academia-to-industry.

[39] TWI Ltd, *Robotic arc welding*. [Online]. Available: https://www.twi-global.com/technical-knowledge/job-knowledge/robotic-arc-welding-135.

[40] T. Bonine, *Robotic welding advantages*, Jul. 2015. [Online]. Available: https://www.automation.com/en-us/articles/2015-2/robotic-welding-advantages.

[41] P. Rocadas and R. McMaster, "A robot cell calibration algorithm and its use with a 3d measuring system," in *ISIE '97 Proceeding of the IEEE International Symposium on Industrial Electronics*, vol. 1, 1997, SS297–SS302 vol.1. DOI: 10.1109/ISIE.1997.651779.

[42] RoboDK Inc., *Robot calibration*. [Online]. Available: https://robodk.com/robot-calibration.

[43] A. Joubair, *What are accuracy and repeatability in industrial robots?* May 2016. [Online]. Available: https://blog.robotiq.com/bid/72766/What-are-Accuracy-and-Repeatability-in-Industrial-Robots.

[44] ABB, *Robotstudio the world's most used offline programming tool for robotics*. [Online]. Available: https://new.abb.com/products/robotics/robotstudio.

[45] KUKA Robotics Corporation, *Kuka.sim*. [Online]. Available: https://www.kuka.com/en-us/products/robotics-systems/software/simulation-planning-optimization/kuka_sim.

[46] AYVA Educational Solutions, *Kuka.sim software*. [Online]. Available: https://www.ayva.ca/eng/product/kuka-sim-software/.

[47] W. Meisen, *Kuka invests in the factory of the future*, Dec. 2017. [Online]. Available: https://www.kuka.com/en-de/press/news/2017/12/kuka-akquiriert-visual-components.

[48] YASKAWA AMERICA, INC., *Motosim eg-vrc*, 2021. [Online]. Available: https://www.motoman.com/getmedia/e94bc0ea-c62a-4977-bc98-fdfe598a490f/MotoSimEG_VRC.pdf.aspx.

[49] S. Cranston, *3 styles of robot programming using offline simulation software (motosim)*, Mar. 2018. [Online]. Available: https://www.linkedin.com/pulse/3-styles-robot-programming-using-offline-simulation-motosim-cranston/.

[50] A. Nubiola, *The future of robot off-line programming*, Dec. 2015. [Online]. Available: https://robohub.org/the-future-of-robot-off-line-programming/.

[51] Autoline Automation Ltd, *Motoopc server software*. [Online]. Available: https://autoline.nz/products/robotic-automation/yaskawa-motoman-robotics/yaskawa-software/motoopc-server-software/.

[52] ABB, *Product specification robotstudio*, ABB, Mar. 2021. [Online]. Available: %5Curl%7Bhttps://library.e.abb.com/public/218ec1270fd24602a5d69fba5d5098
3HAC026932%5C%20PS%5C%20RobotStudio-en.pdf?x-sign=iL0xb7BVVaEgj326mYNVZS
20%7D.

[53] Intelitek, *Motosim*. [Online]. Available: https://intelitek.com/motosim/.

[54] ABB, *Release notes for robotstudio sdk 2021.1*, Mar. 2021. [Online]. Available: https://developercenter.robotstudio.com/api/robotstudio/articles/releases/robotstudio-sdk-2021-1.html#add-documentwindow-example-to-sdk-template.

[55] RoboDK Inc., *Simulate robot applications program any industrial robot with one simulation environment*. [Online]. Available: https://robodk.com/index.

[56] Delfoi Robotics, *Delfoi robotics software*. [Online]. Available: https://www.delfoi.com/delfoi-robotics/delfoi-robotics-software/.

[57] Visual Components, *About us*. [Online]. Available: https://www.visualcomponents.com/about-us/.

[58] ——, *Meet the heads of a simulation software family*, Feb. 2020. [Online]. Available: https://www.visualcomponents.com/about-us/meet-the-team/meet-the-founders-of-visual-components/.

[59] ——, *Steps of the olp process*, Dec. 2017. [Online]. Available: https://www.visualcomponents.com/resources/articles/steps-of-the-olp-process/.

[60] ——, *Supported cad files*. [Online]. Available: https://www.visualcomponents.com/supported-cad-files/.

[61] ——, *Premium*. [Online]. Available: https://www.visualcomponents.com/products/premium/.

[62] ——, *System requirements*. [Online]. Available: https://www.visualcomponents.com/system-requirements/.

[63] RoboDK, *Robodk pricing*. [Online]. Available: https://robodk.com/pricing.

[64] A. Owen-Hill, *9 powerful robodk features you might not know about*, Oct. 2020. [Online]. Available: https://robodk.com/blog/powerful-robodk-features/.

[65] Delfoi Robotics, *System requirements*. [Online]. Available: https://www.delfoi.com/delfoi-robotics/system-requirements/.

[66] YASKAWA ELECTRIC CORPORATION, *What is "robot"*. [Online]. Available: https://www.yaskawa-global.com/product/robotics/about.

[67] M. Bélanger-Barrette, *What is included in robotic welding systems?* Feb. 2016. [Online]. Available: https://blog.robotiq.com/bid/72927/What-is-Included-in-Robotic-Welding-Systems.

[68] Robots Done Right, *The main components of an industrial robot*. [Online]. Available: https://robotsdoneright.com/Articles/main-components-of-an-industrial-robot.html.

[69] Yaskawa Europe GmbH, *Yaskawa motoman robot controllers yrc1000*. [Online]. Available: https://www.yaskawa.eu.com/products/robots/controller/productdetail/product/yrc1000_583.

[70] Yaskawa America, *Robotics glossary*. [Online]. Available: https://www.motoman.com/en-us/about/company/robotics-glossary.

[71] Fronius International GmbH, *Cmt – cold metal transfer: The cold welding process for premium quality*. [Online]. Available: https://www.fronius.com/en/welding-technology/world-of-welding/fronius-welding-processes/cmt.

[72] A. Øvern, "Industry 4.0-digital twins and opc ua," M.S. thesis, NTNU, 2018.

[73] Fronius International GmbH, *Tps/i*. [Online]. Available: https://www.fronius.com/en/welding-technology/products/manual-welding/migmag/tpsi/tpsi/tps-400i.

[74] Yaskawa Europe GmbH, *Yaskawa*. [Online]. Available: https://www.yaskawa.eu.com/products/robots/handling-mounting/productdetail/product/gp25_699.

Appendix A.

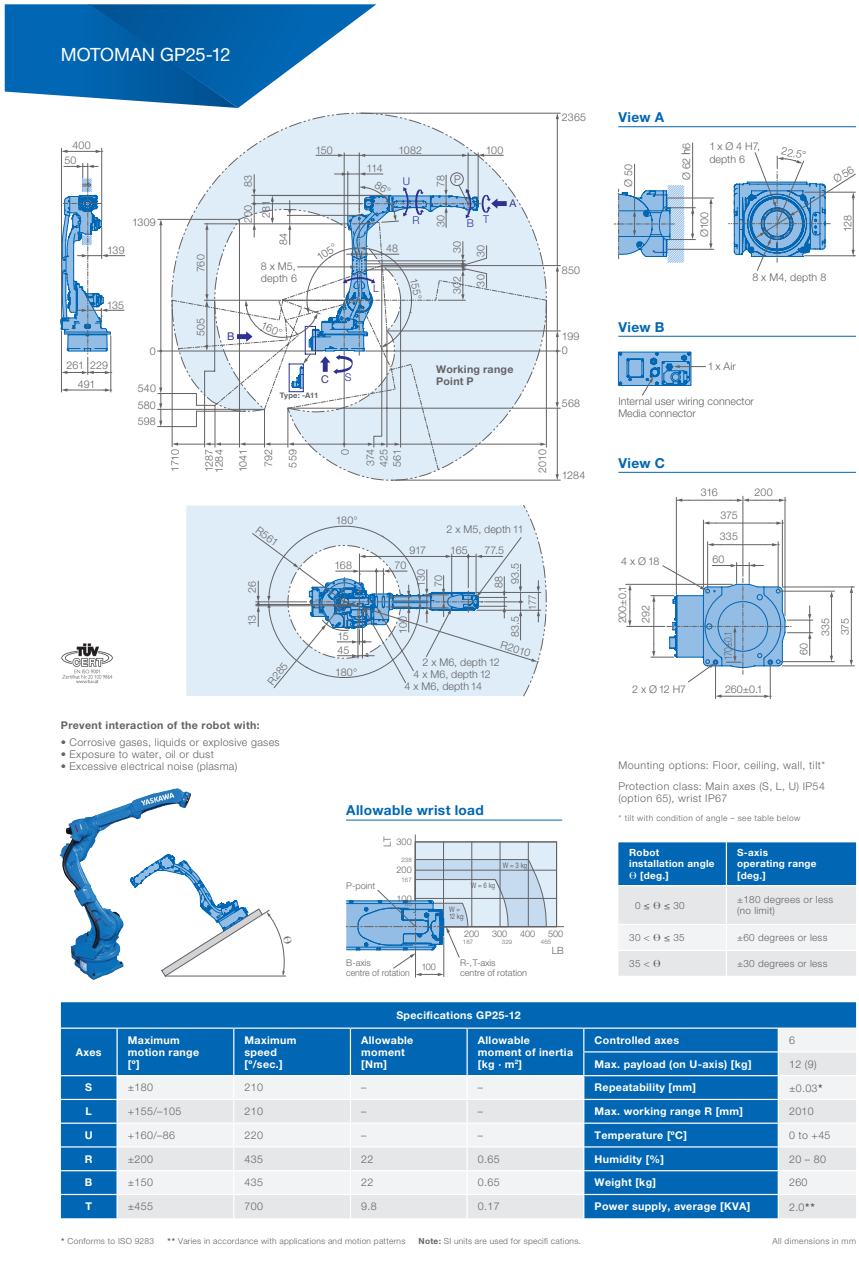
System description of robot cell and Python code

A.1. Fronius TPS 400i

Table A.1.: Technical data for TPS 400i [73]

Welding current max.	400 A
Welding current min.	3 A
Welding current / Duty cycle [10min/40°C]	400A / 40%
Welding current / Duty cycle [10min/40°C]	360A / 60%
Welding current / Duty cycle [10min/40°C]	320A / 100%
Operating voltage	14,2-34,0V
Open-circuit voltage	73 V
Mains frequency	50-60Hz
Mains voltage	3 x 400V
Mains fuse	35A
Dimension / b	300 mm
Dimension / l	706 mm
Weight	36,45 kg
Degree of protection	IP23

A.2. Yaskawa Motoman GP25-12



Robot installation angle Θ [deg.]	S-axis operating range [deg.]
0 ≤ Θ ≤ 30	±180 degrees or less (no limit)
30 < Θ ≤ 35	±60 degrees or less
35 < Θ	±30 degrees or less

Specifications GP25-12						
Axes	Maximum motion range [°]	Maximum speed [°/sec.]	Allowable moment [Nm]	Allowable moment of inertia [kg · m²]	Controlled axes	6
S	±180	210	–	–	Max. payload (on U-axis) [kg]	12 (9)
L	+155/–105	210	–	–	Repeatability [mm]	±0.03*
U	+160/–86	220	–	–	Max. working range R [mm]	2010
R	±200	435	22	0.65	Temperature [°C]	0 to +45
B	±150	435	22	0.65	Humidity [%]	20 – 80
T	±455	700	9.8	0.17	Weight [kg]	260
					Power supply, average [KVA]	2.0**

* Conforms to ISO 9283 ** Varies in accordance with applications and motion patterns **Note:** SI units are used for specifi cations. All dimensions in mm

Figure A.1.: Description of Yaskawa Motoman GP25-12. Source: [74]

Table A.2.: Specifications for Yaskawa Motoman GP25-12 part 1 [74]

Axes	Maximum motion range [°]	Maximum speed [°/sec]	Allowable moment [N·m]	Allowable moment of inertia [kg·m^2]
S	± 180	210	-	-
L	+155/-105	210	-	-
U	+160/-86	220	-	-
R	± 200	435	22	0.65
B	± 150	435	22	0.65
T	± 455	700	9.8	0.17

Table A.3.: Specifications for Yaskawa Motoman GP25-12 part 2 [74]

Controlled axes	6
Max. payload (on U-axis) [kg]	12(9)
Repeatability [mm]	± 0.03*
Max. working range R [mm]	2010
Temperature [°C]	0 to +45
Humidity [%]	20-80
Weight [kg]	260
Power supply, average [KVA]	2,0**

*Conforms to ISO 9283 **Varies in accordance with applications and motion patterns

A.3. Python script for calculating robot kinematics

```
1
2  import numpy as np
3  import modern_robotics as mr
4  import math
5  from numpy.linalg import matrix_rank
6  from IPython import display
7
8  #Defining lengths [mm].
9  l1 = 505
10 l2 = 760
11 l3 = 200
12 l4 = 150
13 l5 = 275
14 l6 = 807
15 l7 = 100
16
17 #Values for x_tool, y_tool, and z_tool gathered from the programming
       pendant
18 x_tool = 0
19 y_tool = 450.27
20 z_tool = 84.4028
21
22 #M matrix for Yaskawa Motoman GP25-12 in zero-position without
       welding gun
23 M = np.array([[1,0,0,0],
24              [0,1,0,l4+l5+l6+l7],
25              [0,0,1,l1+l2+l3],
26              [0,0,0,1]])
27
28
29 #M matrix for Yaskawa Motoman GP25-12 in zero-position with welding
       gun
30 theta_endEffector = np.radians(-54)
31
32 R_endEffector = np.array([[1,0,0],
33              [0,np.cos(theta_endEffector ),-np.sin(theta_endEffector
       )],
34              [0,np.sin(theta_endEffector ), np.cos(theta_endEffector)
       ]])
35
36 p_endEffector = np.array([x_tool,y_tool,z_tool])
37
38 M_tool = mr.RpToTrans(R_endEffector,p_endEffector)
39
40 #Calculating each screw axis in space form Sn
41 w1 = np.array([0,0,1])
42 q1 = np.array([0,0,0])
43 v1 = np.cross(-w1,q1)
44 S1 = np.append(w1,v1)
45
```

```
w2 = np.array([1,0,0])
q2 = np.array([0,l4,l1])
v2 = np.cross(-w2,q2)
S2 = np.append(w2,v2)

w3 = np.array([1,0,0])
q3 = np.array([0,l4,l1+l2])
v3 = np.cross(-w3,q3)
S3 = np.append(w3,v3)

w4 = np.array([0,1,0])
q4 = np.array([0,l4+l5,l1+l2+l3])
v4 = np.cross(-w4,q4)
S4 = np.append(w4,v4)

w5 = np.array([1,0,0])
q5 = np.array([0,l4+l5+l6,l1+l2+l3])
v5 = np.cross(-w5,q5)
S5 = np.append(w5,v5)

w6 = np.array([0,1,0])
q6 = np.array([0,l4+l5+l6+l7,l1+l2+l3])
v6 = np.cross(-w6,q6)
S6 = np.append(w6,v6)

Slist = np.concatenate(([S1], [S2], [S3], [S4], [S5], [S6]), axis=0)
    .T

#Calculating each screw matrix [Sn]
se1 = mr.VecTose3(S1)
se2 = mr.VecTose3(S2)
se3 = mr.VecTose3(S3)
se4 = mr.VecTose3(S4)
se5 = mr.VecTose3(S5)
se6 = mr.VecTose3(S6)

#Calculating each screw axis in body form Bn
wb1 = np.array([0,0,1])
qb1 = np.array([0,-(l7+l6+l5+l4),-(l3+l2+l1)])
vb1 = np.cross(-wb1,qb1)
Sb1 = np.append(wb1,vb1)

wb2 = np.array([1,0,0])
qb2 = np.array([0,-(l7+l6+l5),-(l3+l2)])
vb2 = np.cross(-wb2,qb2)
Sb2 = np.append(wb2,vb2)

wb3 = np.array([1,0,0])
qb3 = np.array([0,-(l7+l6+l5),-l3])
vb3 = np.cross(-wb3,qb3)
Sb3 = np.append(wb3,vb3)
```

```
96
97  wb4 = np.array([0,1,0])
98  qb4 = np.array([0,-(l7+l6),0])
99  vb4 = np.cross(-wb4,qb4)
100 Sb4 = np.append(wb4,vb4)
101
102 wb5 = np.array([1,0,0])
103 qb5 = np.array([0,-l7,0])
104 vb5 = np.cross(-wb5,qb5)
105 Sb5 = np.append(wb5,vb5)
106
107 wb6 = np.array([0,1,0])
108 qb6 = np.array([0,0,0])
109 vb6 = np.cross(-wb6,qb6)
110 Sb6 = np.append(wb6,vb6)
111
112 Blist = np.concatenate(([Sb1], [Sb2], [Sb3], [Sb4], [Sb5], [Sb6]),
        axis=0).T
113
114 #Defining each joint angle
115 theta1 = -np.pi/2
116 theta2 = -np.pi/5
117 theta3 = 0
118 theta4 = 0
119 theta5 = -np.pi/2
120 theta6 = 0
121
122 #Calculating the forward kinematics for the Yaskawa Motoman GP25-12
123
124 thetalist = np.array([theta1,theta2,theta3,theta4,theta5, theta6])
125
126 T = mr.FKinSpace(M_tool,Slist,thetalist)
127
128 T2 = mr.FKinBody(M_tool,Blist,thetalist)
129
130 #Calculating the inverse kinematics for the Yaskawa Motoman GP25-12
131
132 #Defining the desired end-effector configuration to be equal to the
        solution of the forward kinematics
133 T_goal1 = T
134 T_goal2 = T
135
136 #Defining a guess for each joint angle, as well as an positive error
         allowance
137 thetalist_guess = np.array([-1,-0.3,0,0,-1.2,0])
138 print('This is a list of guessed angles sufficiently close to the
        solution. The result should be "true"')
139 print(thetalist_guess)
140
141 thetalist_guess2 = np.array([-25,5,1,1,-1.2,2])
142 print('This is a list of guessed angles to far from the solution.
```

```
    The result should be "false"')
print(thetalist_guess2)

eomg = 0.001
ev = 0.0001

#Calculating the inverse kinematics for the Yaskawa Motoman GP25-12

[thetalist_01,result1] = mr.IKinSpace(Slist,M_tool,T_goal1,
    thetalist_guess,eomg, ev)
print(' "Solution to inverse kinematics where initial guess of joint
     angles sufficiently close to solution"')
print(thetalist_01, result1)

[thetalist_02,result2] = mr.IKinSpace(Slist,M_tool,T_goal2,
    thetalist_guess2,eomg, ev)
print("Solution to inverse kinematics where initial guess of joint
    angles too far from solution")
print(thetalist_02, result2)
```

Listing A.1: Python code for calculating robot kinematics

Appendix B.

Testing OPC UA connection

```
1
2  import sys
3  import socket
4  sys.path.insert(0, "..")
5  import time
6  from moto.simple_message import *
7  from moto import motion_connection
8  from moto.simple_message_connection import SimpleMessageConnection
9  from moto.simple_message import SimpleMessage
10 from moto import Moto
11 from moto import Moto, ControlGroupDefinition
12 from moto.simple_message import (
13     JointFeedbackEx,
14     JointTrajPtExData,
15     JointTrajPtFull,
16     JointTrajPtFullEx,
17 )
18
19 from opcua import ua, Server
20 import numpy as np
21
22 robot: Moto = Moto(
23         "192.168.255.200",
24         [ControlGroupDefinition("robot", 0, 6, ["s", "l", "u", "r",
       "b", "t"])],
25     )
26
27 robot.motion.start_servos()
28 robot.motion.start_trajectory_mode()
29
30 time.sleep(1)
31
32 status = robot.motion.check_motion_ready()
33 print(status)
34
```

```
position = robot.state.joint_feedback(0).pos
print("------------------joint position")
print(position)

def disconnect(self):
    client.close()

if __name__ == "__main__":

    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # setup our server
    server = Server()
    server.set_endpoint("opc.tcp://'localhost':4840/motopcua/server/
    ")

    # setup our own namespace, not really necessary but should as
    spec
    uri = "http://server.motopcua.github.io"
    idx = server.register_namespace(uri)

    # get Objects node, this is where we should put our nodes
    objects = server.get_objects_node()

    # populating our address space
    myobj = server.nodes.objects.add_object(idx, "Moto")
    myvar = myobj.add_variable(idx, "MyVariable", 6.7)

    myvar.set_writable()    # Set MyVariable to be writable by
    clients

    # Adding desired variables
    s = myobj.add_variable(idx, "s", 6.7)
    l = myobj.add_variable(idx, "l", 6.7)
    u = myobj.add_variable(idx, "u", 6.7)
    r = myobj.add_variable(idx, "r", 6.7)
    b = myobj.add_variable(idx, "b", 6.7)
    t = myobj.add_variable(idx, "t", 6.7)

    s.set_writable()
    l.set_writable()
    u.set_writable()
    r.set_writable()
    b.set_writable()
    t.set_writable()

    # starting!
    server.start()
    print("server is started")
    try:

```

```
83          while True:
84              time.sleep(0.1)
85
86              s.set_value(np.rad2deg(position[0]))
87              l.set_value(np.rad2deg(position[1]))
88              u.set_value(np.rad2deg(position[2]))
89              r.set_value(np.rad2deg(position[3]))
90              b.set_value(np.rad2deg(position[4]))
91              t.set_value(np.rad2deg(position[5]))
92
93      finally:
94          #close connection, remove subcsriptions, etc
95          server.stop()
```

Listing B.1: OPC UA server reading joint positions from physical robot and writing these to Visual Components

Appendix C.

The finalized Python scripts

C.1. first attempt at sending trajectory points to robot controller

```
1
2  import sys
3  import csv
4  import socket
5  sys.path.insert(0, "..")
6  import time
7  from moto.simple_message import *
8  from moto import motion_connection
9  from moto.simple_message_connection import SimpleMessageConnection
10 from moto.simple_message import SimpleMessage
11 from moto import Moto
12 from moto import Moto, ControlGroupDefinition
13 from moto.simple_message import (
14     JointFeedbackEx,
15     JointTrajPtExData,
16     JointTrajPtFull,
17     JointTrajPtFullEx,
18 )
19
20 from opcua import ua, Server
21 import numpy as np
22 import json
23
24 def get_joint_values_from_file(filepath, variable_name):
25
26     data = json.load(open(filepath))
27
28     return np.asarray(data[variable_name])
29
30 robot: Moto = Moto(
31         "192.168.255.200",
```

```
        [ControlGroupDefinition("robot", 0, 6, ["s", "l", "u", "r", "b", "t"])],
    )
robot.motion.start_servos()
robot.motion.start_trajectory_mode()
time.sleep(1)
status = robot.motion.check_motion_ready()
print("----------------------status of robot ready/not ready?: ")
print(status)

position = robot.state.joint_feedback(0).pos
print("----------------------current position of robot joints: ")
print(position)

home = JointTrajPtFull(
                    groupno=0,
                    sequence=0,
                    valid_fields=ValidFields.TIME | ValidFields.POSITION | ValidFields.VELOCITY,
                    time=0,
                    pos = [0.0]*10,
                    vel = [0.0]*10,
                    acc = [0.0]*10
                )

threshold = 0.003
check = list(np.array(position)-np.array(home.pos))
print("----check",check)

#check whether or not robot is in home position
for item in check:
    if item > threshold:

        p0 = JointTrajPtFull(
                        groupno=0,
                        sequence=0,
                        valid_fields=ValidFields.TIME | ValidFields.POSITION | ValidFields.VELOCITY,
                        time=0,
                        pos = position,
                        vel = [0.0]*10,
                        acc = [0.0]*10
                    )
        robot.motion.send_joint_trajectory_point(p0)
        time.sleep(1)
        home = JointTrajPtFull(
                        groupno=0,
                        sequence=1,
                        valid_fields=ValidFields.TIME | ValidFields.POSITION | ValidFields.VELOCITY,
                        time=5.0,
```

```
                        pos = [0.0]*10,
                        vel = [0.0]*10,
                        acc = [0.0]*10
                    )
        robot.motion.send_joint_trajectory_point(home)
        time.sleep(6)
        p0 = JointTrajPtFull(
                        groupno=0,
                        sequence=0,
                        valid_fields=ValidFields.TIME | ValidFields.
    POSITION | ValidFields.VELOCITY,
                        time=0,
                        pos = robot.state.joint_feedback(0).pos,
                        vel = [0.0]*10,
                        acc = [0.0]*10
                    )
        robot.motion.send_joint_trajectory_point(p0)

        print("robot has been moved to home pos and p0 updated", p0.
    pos)

p0 = JointTrajPtFull(
                groupno=0,
                sequence=0,
                valid_fields=ValidFields.TIME | ValidFields.POSITION
     | ValidFields.VELOCITY,
                time=0,
                pos = position,
                vel = [0.0]*10,
                acc = [0.0]*10
            )
robot.motion.send_joint_trajectory_point(p0)
print("robot was in home pos, p0 updated")

TrajPoints_fromSim = get_joint_values_from_file(filepath="
    test_verdier_joint.json", variable_name="joint_values")

for row in TrajPoints_fromSim:

    zero_array = np.array([0,0,0,0])
    traj_point = np.append(row, zero_array)

    p0 = JointTrajPtFull(
                    groupno=0,
                    sequence=0,
                    valid_fields=ValidFields.TIME | ValidFields.
    POSITION | ValidFields.VELOCITY,
                    time=0,
                    pos = robot.state.joint_feedback(0).pos,
                    vel = [0.0]*10,
                    acc = [0.0]*10
```

```
125                 )
126     robot.motion.send_joint_trajectory_point(p0)
127     print(p0.pos)
128     time.sleep(1)
129 
130     p1 = JointTrajPtFull(
131                     groupno=0,
132                     sequence=1,
133                     valid_fields=ValidFields.TIME | ValidFields.
    POSITION | ValidFields.VELOCITY,
134                     time=20,
135                     pos = traj_point,
136                     vel = [0.0]*10,
137                     acc = [0.0]*10
138                 )
139     robot.motion.send_joint_trajectory_point(p1)
140     print(p1.pos)
141     time.sleep(20)
```

Listing C.1: First attempt at sending joint position vectors from file to physical robot

C.2. Python script for reading from simulation and writing to robot controller simultaneously

```
import sys
import csv
import socket
sys.path.insert(0, "..")
import time
from moto.simple_message import *
from moto import motion_connection
from moto.simple_message_connection import SimpleMessageConnection
from moto.simple_message import SimpleMessage
from moto import Moto
from moto import Moto, ControlGroupDefinition
from moto.simple_message import (
    JointFeedbackEx,
    JointTrajPtExData,
    JointTrajPtFull,
    JointTrajPtFullEx,
)
from opcua import ua, Server
import numpy as np
import json

robot_pos_file = "robot_pos.json"
simulation_pos_file = "sim_pos.json"

robot: Moto = Moto(
        "192.168.255.200",
        [ControlGroupDefinition("robot", 0, 6, ["s", "l", "u", "r",
    "b", "t"])],
    )

#connect to physical robot and prepare for movement
robot.motion.start_servos()
robot.motion.start_trajectory_mode()
check_ready = robot.motion.check_motion_ready()
print("robot ready?.............")
print(check_ready)
time.sleep(3)

def disconnect(self):
    client.close()

if __name__ == "__main__":

    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # setup our server
    server = Server()
    server.set_endpoint("opc.tcp://localhost:4840/motopcua/server/")
```

```
48
49      # setup our own namespace, not really necessary but should as
        spec
50      uri = "http://server.motopcua.github.io"
51      idx = server.register_namespace(uri)
52
53      # get Objects node, this is where we should put our nodes
54      objects = server.get_objects_node()
55
56      # populating our address space
57      myobj = server.nodes.objects.add_object(idx, "Moto")
58      myvar = myobj.add_variable(idx, "MyVariable", 6.7)
59
60      myvar.set_writable()    # Set MyVariable to be writable by
        clients
61
62      # Adding desired variables from server to simulation
63      s = myobj.add_variable(idx, "s", 6.7)
64      l = myobj.add_variable(idx, "l", 6.7)
65      u = myobj.add_variable(idx, "u", 6.7)
66      r = myobj.add_variable(idx, "r", 6.7)
67      b = myobj.add_variable(idx, "b", 6.7)
68      t = myobj.add_variable(idx, "t", 6.7)
69
70      # set joints writable
71      s.set_writable()
72      l.set_writable()
73      u.set_writable()
74      r.set_writable()
75      b.set_writable()
76      t.set_writable()
77
78      #Adding desired variables from simulation to server
79      joint_S = myobj.add_variable(idx, "S1", 6.7)
80      joint_L = myobj.add_variable(idx, "L1", 6.7)
81      joint_U = myobj.add_variable(idx, "U1", 6.7)
82      joint_R = myobj.add_variable(idx, "R1", 6.7)
83      joint_B = myobj.add_variable(idx, "B1", 6.7)
84      joint_T = myobj.add_variable(idx, "T1", 6.7)
85
86      joint_S.set_writable()
87      joint_L.set_writable()
88      joint_U.set_writable()
89      joint_R.set_writable()
90      joint_B.set_writable()
91      joint_T.set_writable()
92
93      # starting!
94      server.start()
95      print("server is started")
96      try:
```

```
97          pos_vec_fromRob = []
98          pos_vec = []
99          index = 0
100         timer = 10
101         while True:
102
103             S_updated = joint_S.get_value()
104             L_updated = joint_L.get_value()
105             U_updated = joint_U.get_value()
106             R_updated = joint_R.get_value()
107             B_updated = joint_B.get_value()
108             T_updated = joint_T.get_value()
109
110             if S_updated != 6.7 or L_updated != 6.7 or U_updated !=
    6.7 or R_updated != 6.7 or B_updated != 6.7 or T_updated != 6.7:
111                 vec = [S_updated, L_updated, U_updated, R_updated,
    B_updated, T_updated]
112
113                 data = json.load(open(simulation_pos_file))
114                 data_vec = data["joint_values"]
115                 data_vec.append(vec)
116                 data["joint_values"] = data_vec
117                 with open(simulation_pos_file, "w") as f:
118                     json.dump(data, f, indent=2)
119
120                 pos_vec.append(vec)
121                 print(pos_vec)
122
123                 position = pos_vec[index]
124
125                 zero_array = np.array([0,0,0,0])
126                 trajectory_point = list(np.append(position,
    zero_array))
127
128                 if index == 0:
129                     currentPosition = JointTrajPtFull(
130                             groupno=0,
131                             sequence=0,
132                             valid_fields=ValidFields.TIME |
    ValidFields.POSITION | ValidFields.VELOCITY,
133                             time=0,
134                             pos = robot.state.joint_feedback(0).pos,
135                             vel = [0.0]*10,
136                             acc = [0.0]*10
137                         )
138                     robot.motion.send_joint_trajectory_point(
    currentPosition)
139
140                     time.sleep(0.1)
141                     newPosition = JointTrajPtFull(
142                             groupno=0,
```

```
                            sequence=index+1,
                            valid_fields=ValidFields.TIME |
    ValidFields.POSITION | ValidFields.VELOCITY,
                            time=timer,
                            pos = [0.0]*10,
                            vel = [0.0]*10,
                            acc = [0.0]*10
                        )
                    robot.motion.send_joint_trajectory_point(
    newPosition)
                    time.sleep(0.2)
                    index = 1
                    timer +=0.2

                else:

                    newPosition = JointTrajPtFull(
                                groupno=0,
                                sequence=index+1,
                                valid_fields=ValidFields.TIME |
    ValidFields.POSITION | ValidFields.VELOCITY,
                                time=timer,
                                pos = np.deg2rad(trajectory_point),
                                vel = [0.0]*10,
                                acc = [0.0]*10
                        )
                    robot.motion.send_joint_trajectory_point(
    newPosition)

                    time.sleep(0.3)

                    position2 = robot.state.joint_feedback(0).pos
                    rad = np.rad2deg(position2)

                    s.set_value(rad[0])
                    l.set_value(rad[1])
                    u.set_value(rad[2])
                    r.set_value(rad[3])
                    b.set_value(rad[4])
                    t.set_value(rad[5])

                    position_vector = robot.state.joint_feedback(0).
    pos
                    s_from_rob = position_vector[0]
                    l_from_rob = position_vector[1]
                    u_from_rob = position_vector[2]
                    r_from_rob = position_vector[3]
                    b_from_rob = position_vector[4]
                    t_from_rob = position_vector[5]

                    vec_fromRob = [s_from_rob, l_from_rob,
```

```
    u_from_rob, r_from_rob, b_from_rob, t_from_rob]
                    data2 = json.load(open(robot_pos_file))
                    data_vec2 = data2["joint_values"]
                    data_vec2.append(vec_fromRob)
                    data2["joint_values"] = data_vec2

                    with open(robot_pos_file, "w") as f:
                        json.dump(data2, f, indent=2)
                    pos_vec_fromRob.append(vec_fromRob)

                    index += 1
                    timer += 0.2

    finally:
        #close connection, remove subcsriptions, etc
        server.stop()
```

Listing C.2: Script that reads from simulation and directly sends to physical robot

C.3. OPC UA server writing trajectory from simulation to file

```
import sys
import csv
import socket
sys.path.insert(0, "..")
import time
from moto.simple_message import *
from moto import motion_connection
from moto.simple_message_connection import SimpleMessageConnection
from moto.simple_message import SimpleMessage
from moto import Moto
from moto import Moto, ControlGroupDefinition
from moto.simple_message import (
    JointFeedbackEx,
    JointTrajPtExData,
    JointTrajPtFull,
    JointTrajPtFullEx,
)
import copy
from opcua import ua, Server
import numpy as np
import json

def trajectory_continuing(pos_vec, history_vec):

    history_vec.append(pos_vec)

```

```
    if len(history_vec) <= 4:
        continuing = True
        return history_vec, continuing

    else:

        temp = np.asarray(history_vec[-4:])
        print(temp)
        print(temp.shape)

        if sum(abs(temp[-1])-abs(temp[-2])) < 10**-3 and sum(abs(temp[-1])-abs(temp[-3])) < 10**-3 and sum(abs(temp[-1])-abs(temp[-4])) < 10**-3:
            continuing = False
            return history_vec, continuing

        else:
            continuing = True
            return history_vec, continuing

def disconnect(self):
    client.close()

if __name__ == "__main__":

    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # setup our server
    server = Server()
    server.set_endpoint("opc.tcp://localhost:4840/motopcua/server/")

    # setup our own namespace, not really necessary but should as spec
    uri = "http://server.motopcua.github.io"
    idx = server.register_namespace(uri)

    # get Objects node, this is where we should put our nodes
    objects = server.get_objects_node()

    # populating our address space
    myobj = server.nodes.objects.add_object(idx, "Moto")
    myvar = myobj.add_variable(idx, "MyVariable", 6.7)

    myvar.set_writable()    # Set MyVariable to be writable by clients

    # Adding desired variables
    s = myobj.add_variable(idx, "s", 6.7)
    l = myobj.add_variable(idx, "l", 6.7)
    u = myobj.add_variable(idx, "u", 6.7)
    r = myobj.add_variable(idx, "r", 6.7)
```

```
    b = myobj.add_variable(idx, "b", 6.7)
    t = myobj.add_variable(idx, "t", 6.7)

    s.set_writable()
    l.set_writable()
    u.set_writable()
    r.set_writable()
    b.set_writable()
    t.set_writable()


    # starting!
    server.start()
    print("server is started")
    try:
        pos_vec = []
        history_vec = []
        data_vec = []
        time_vec = []
        count = 1

        filename = "serverDouble_sin_test4.json"
        start_time = time.time()
        while True:

            time.sleep(0.2)
            new_s = s.get_value()
            new_l = l.get_value()
            new_u = u.get_value()
            new_r = r.get_value()
            new_b = b.get_value()
            new_t = t.get_value()

            #Start writing to file when simulation is started in
    Visual Components
            if new_s != 6.7 or new_l != 6.7 or new_u != 6.7 or new_r
     != 6.7 or new_b != 6.7 or new_t != 6.7:

                prev_time = start_time

                if count > 1:
                    time_var = prev_time-start_time
                    time_vec.append(time_var)

                start_time = time.time()

                vec = [new_s, new_l, new_u, new_r, new_b, new_t]

                history_vec, continuing = trajectory_continuing(
    pos_vec=vec, history_vec=history_vec)
                #if new_s == new_s-1 and new_l == new_l-1 etc  s
```

```
    m   den vente
                if continuing:

                    data = json.load(open(filename))

                    if count != 1:
                        data_vec = data["joint_values"]

                    data_vec.append(vec)

                    data["joint_values"] = data_vec
                    #print(type(data))
                    #print(data)

                    with open(filename, "w") as f:
                        json.dump(data, f, indent=2)

                    pos_vec.append(vec)
                else:
                    time_vec_final = time_vec[0:-1]

                count += 1

                if count>3 and new_s == 0.0 and new_l == 0.0 and
    new_u == 0.0 and new_r == 0.0 and new_b == 0.0 and new_t == 0.0:
                    print("tajectory finished")
                    server.stop()

    finally:
        #close connection, remove subcsriptions, etc
        server.stop()
```

Listing C.3: OPC UA server reading joint positions from Visual Components and writing these to file

C.4. Final script for sending trajectory from file to robot controller

```

import sys
import socket
sys.path.insert(0, "..")
import time
from moto.simple_message import *
from moto import motion_connection
from moto.simple_message_connection import SimpleMessageConnection
from moto.simple_message import SimpleMessage
from moto import Moto
from moto import Moto, ControlGroupDefinition
```

```
from moto.simple_message import (
    JointFeedbackEx,
    JointTrajPtExData,
    JointTrajPtFull,
    JointTrajPtFullEx,
)
import copy
from opcua import ua, Server
import numpy as np
import json
from simtorob import SimtoRob

def get_joint_values_from_file(filepath, variable_name):

    data = json.load(open(filepath))

    return np.asarray(data[variable_name])

pos_from_simulation = "serverDouble_sim_test4.json"
Pos_from_robot = "serverDouble_robot_test4.json"

robot: SimtoRob = SimtoRob(
        "192.168.255.200",
        [ControlGroupDefinition("robot", 0, 6, ["s", "l", "u", "r", "b", "t"])],
    )

robot.motion.start_servos()
robot.motion.start_trajectory_mode()

time.sleep(1)

status = robot.motion.check_motion_ready()
print("----------------------status of robot ready/not ready?: ")
print(status)

position = robot.state.joint_feedback(0).pos
print("----------------------current position of robot joints: ")
print(position)

home = JointTrajPtFull(
                    groupno=0,
                    sequence=0,
                    valid_fields=ValidFields.TIME | ValidFields.POSITION | ValidFields.VELOCITY,
                    time=0,
                    pos = [0.0]*10,
                    vel = [0.0]*10,
                    acc = [0.0]*10
                )

```

```
61 threshold = 0.00003
62
63 check = list(np.array(position)-np.array(home.pos))
64 print("----check",check)
65
66 for item in check:
67
68     sequence_nb = 0
69
70     if abs(item) > threshold:
71         timer = 5.0
72         p0 = JointTrajPtFull(
73                         groupno=0,
74                         sequence=0,
75                         valid_fields=ValidFields.TIME | ValidFields.
    POSITION | ValidFields.VELOCITY,
76                         time=0,
77                         pos = position,
78                         vel = [0.0]*10,
79                         acc = [0.0]*10
80                     )
81         robot.motion.send_joint_trajectory_point(p0)
82         time.sleep(1)
83         home = JointTrajPtFull(
84                         groupno=0,
85                         sequence=sequence_nb+1,
86                         valid_fields=ValidFields.TIME | ValidFields.
    POSITION | ValidFields.VELOCITY,
87                         time=timer,
88                         pos = [0.0]*10,
89                         vel = [0.0]*10,
90                         acc = [0.0]*10
91                     )
92         robot.motion.send_joint_trajectory_point(home)
93         sequence_nb +=1
94         timer +=0.2
95         time.sleep(5)
96         print("robot has been moved to home pos and p0 updated",
    home.pos)
97         break
98
99     else:
100         timer = 0
101         home = JointTrajPtFull(
102                         groupno=0,
103                         sequence=sequence_nb,
104                         valid_fields=ValidFields.TIME | ValidFields.
    POSITION | ValidFields.VELOCITY,
105                         time=timer,
106                         pos = robot.state.joint_feedback(0).pos,
107                         vel = [0.0]*10,
```

```
                        acc = [0.0]*10
                    )
        robot.motion.send_joint_trajectory_point(home)
        sequence_nb +=1
        timer += 0.2

TrajPoints_fromSim = get_joint_values_from_file(filepath="
    serverDouble_sin_test4.json", variable_name="joint_values")

for row in TrajPoints_fromSim:
    pos_vec = []
    zero_array = np.array([0,0,0,0])
    traj_point = np.append(row, zero_array)

    p1 = JointTrajPtFull(
                    groupno=0,
                    sequence=sequence_nb+1,
                    valid_fields=ValidFields.TIME | ValidFields.
    POSITION | ValidFields.VELOCITY,
                    time=timer,
                    pos = np.deg2rad(traj_point),
                    vel = [0.0]*10,
                    acc = [0.0]*10
                )
    robot.motion.send_joint_trajectory_point(p1)
    sequence_nb +=1
    timer += 0.2
    time.sleep(0.2)

    position_vector = robot.state.joint_feedback(0).pos

    new_s = position_vector[0]
    new_l = position_vector[1]
    new_u = position_vector[2]
    new_r = position_vector[3]
    new_b = position_vector[4]
    new_t = position_vector[5]

    vec = [new_s, new_l, new_u, new_r, new_b, new_t]
    data = json.load(open(Pos_from_robot))
    data_vec = data["joint_values"]
    data_vec.append(vec)
    data["joint_values"] = data_vec

    with open(Pos_from_robot, "w") as f:
        json.dump(data, f, indent=2)

    pos_vec.append(vec)
```

Listing C.4: Final script sending offline programmed trajectory to robot controller

Thea Holmedal

Implementing robotic offline programming with the Yaskawa Motoman GP25-12

NTNU
Norwegian University of
Science and Technology