
# Master's Thesis

## Kinematic and dynamic modeling of the Robot ABB IRB 140 for the implementation of control algorithms

| | |
|---|---|
| **Author** | **José Antonio Aliaga Rodríguez** |
| **Master's Degree** | Master's Degree in Industrial Engineering |
| **Modality** | Research Work |
| **University** | University of Almería |
| **Department** | Department of Engineering |
| **Research Group** | ARM-TEP197<br>(Automatic Control, Robotics and Mechatronics) |
| **Date** | September 2019 |

**Director** | **Author**

José Luis Torres Moreno | José Antonio Aliaga Rodríguez

# Abbreviations and acronyms

| | |
|---|---|
| $\vec{p}$ | Pose vector |
| $x$ | Pose component - Location in axis $x$ |
| $y$ | Pose component - Location in axis $y$ |
| $z$ | Pose component - Location in axis $z$ |
| $\vec{q}$ | Quaternion |
| $a$ (*) | Pose component - Quaternion component $a$ |
| $b$ | Pose component - Quaternion component $b$ |
| $c$ | Pose component - Quaternion component $c$ |
| $d$ | Pose component - Quaternion component $d$ |
| $\vec{\tau}$ | Torque vector |
| $D-H$ | Denavit-Hartenberg |
| $\theta$ | Denavit-Hartenberg parameter – Rotation around axis $z$ |
| $d$ | Denavit-Hartenberg parameter – Translation along axis $z$ |
| $a$ (*) | Denavit-Hartenberg parameter – Translation along axis $x$ |
| $\alpha$ | Denavit-Hartenberg parameter – Rotation around axis $x$ |
| $PID$ | Proportional-Integral-Derivative (Controller) |
| $CTC$ | Computed Torque Controller |
| $\vec{q}$ | Joint angles vector |
| $cur$ | Current |
| $des$ | Desired |
| $N$ | Newton |
| $m$ | Meter |
| $s$ | Second |
| $kg$ | Kilogram |
| $rad$ | Radian |

***: Duplicated denominations.**

# Table of contents

# Abstract

With the imminent widespread implementation of Industry 4.0, the field of industrial robotics is on the rise. A much wider presence of new robots in factories, warehouses and workshops connected in a continuous and intelligent way with the rest of the productive elements will undoubtedly be necessary to increase the effectiveness of the industry.

This Master's Thesis presents a real engineering problem that deals with one of the most well-known robots in the industrial robotics landscape: the ABB IRB 140. Programmed both on MATLAB and on its simulation environment Simulink, equivalent kinematic and dynamic models of this robot arm are developed from its real geometric and mechanical features to later implement them in closed loop control schematics.

The main goal of the construction of said control schematics is the positioning and orientation of the end effector of the robot arm in a series of poses along a predefined trajectory very similar to those a real robot would encounter in a practical environment.

In general terms, a complete vision of all the basic steps of the modeling of an articulated robot is implemented, with all necessary scripts for its execution specifically programmed without making use of external robotics libraries.

# Resumen

Con la inminente implementación generalizada de la Industria 4.0, el campo de la robótica industrial se encuentra en alza. Una presencia mucho más amplia de nuevos robots conectados de manera continua e inteligente con el resto de elementos productivos en fábricas, almacenes y talleres será indudablemente necesaria para mejorar la efectividad de la industria.

Este Trabajo Fin de Máster presenta un problema real de ingeniería que trata con uno de los robots más conocidos del panorama de la robótica industrial: el ABB IRB 140. Se desarrollan tanto en MATLAB como en su entorno de simulación Simulink unos modelos cinemático y dinámico equivalentes de este brazo robot a partir de sus características geométricas y mecánicas reales, para posteriormente implementarlos en una serie de esquemas de control en lazo cerrado.

El fin principal de la construcción de dichos esquemas de control es el posicionamiento y orientación del efector final del brazo robot en una serie de poses determinadas a lo largo de una trayectoria predefinida muy similar a las que se vería sometido un robot real en un entorno de trabajo práctico.

En términos generales se proporciona una visión completa de todos aquellos pasos básicos para el modelado de un robot articulado, habiendo sido programados específicamente todos aquellos scripts necesarios para su ejecución sin hacer uso de librerías de robótica externas.

# Chapter 1. Interest, objectives and temporal planning

This chapter presents the motivation for this master's thesis to be written, the main reasons why the topic was selected and in which objectives the whole work was divided.

## 1.1 Historical basis

Throughout history, mankind has been fascinated by any mechanism that could mimic the behavior of living beings. The Greeks had a word for these: *authomatos*, from which the English word automaton derives. Then the Arabs inherited all this knowledge from the ancient world and transmitted it to Europe. It was not until the beginning of the first industrial revolution in the 18th century, when these automatons were design in order to ease the labor to workers and boost production, instead of just amusement.

The word **robot** was introduced for the first time in 1921, when the Czech writer Karel Čapek released in the Prague National Theater his play *Rossum's Universal Robot*. The ethymology of the word is the Czech verb *robota*, which means forced labor, compulsory service or drudgery.

The first modern industrial robot would not appear until 1954, when George C. Devol developed a mechanical arm with a gripper at its end effector, mounted on a rotating platform. Devol, along with Joseph Engelberger, founded the first robotics company in 1961: *Unimation*, and sold this robot to manufacturing companies such as General Motors. It was the beginning of a new industry and since then, robot presence has undergone a continuous and exponential growth spreading to countless production processes, being a key part in enhanced assembly lines.

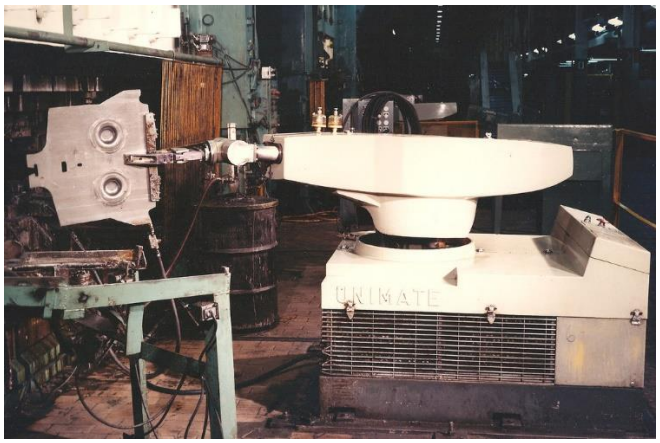



*Figure 1-1. The first robot, developed by Unimation.*

## 1.2 Robotics today

Quality control, painting, assembly, packaging, palletizing (Figure 1-2) or welding are some of the many tasks that an industrial robot can perform, all of them with a degree of precision, repeatability and productivity never achieved before. In addition, this has freed human workers from tedious, repetitive and sometimes hazardous handwork, shifting part of them in the process to more qualified jobs.


*Figure 1-2. Palletizer robot by Bastian Solutions.*

The use of industrial robots has skyrocketed specially in the last decade, reaching 387,000 sales in 2017 and a sales boom is expected in 2019 that will reach 1,400,000 units according to the International Federation of Robotics (IFR). The total estimated number of operational robots by year can be observed in the following Figure 1-3.


*Figure 1-3. Estimated worldwide operational stock of industrial robots. Source: IFR*

Robotics is nowadays one of the leading areas of study in the emerging so-called Industry 4.0, which is a revolution itself in the way we as societies manufacture. Although it was already holding a key role in the previous Industry 3.0, with the advent of new fields such as Machine Learning, the Internet of Things, the cloud or 3D printing, robotics has gained plenty of space for expansion and enhancement.

Among these new possibilities, the reduction of robot prices, the apparition of highly advanced AGVs (Auto-Guided Vehicles) and better features and performance in general.

## 1.3 The ABB IRB 140

The studied robot in this master's thesis is the ABB IRB 140, designed by the swiss company ABB. It is an industrial robot, which by definition of the International Federation of Robotics is a "automatically controlled, reprogrammable multipurpose manipulator programmable in three or more axes". Designed specifically for manufacturing industries, it has an open structure specially adapted for flexible use and can communicate with external systems.

According to the data sheet of the robot, it can handle payloads up to 6 kg with a reach of 810 mm. It can be floor mounted, inverted or wall mounted at any angle and counts with an anti-collision feature to ensure the safety and reliability of performance [1].

This specific model was selected, due to its presence in the laboratories of the University of Almería. This master's thesis is aimed as well to be a bedrock for new degree and master's theses that make use of the robot.

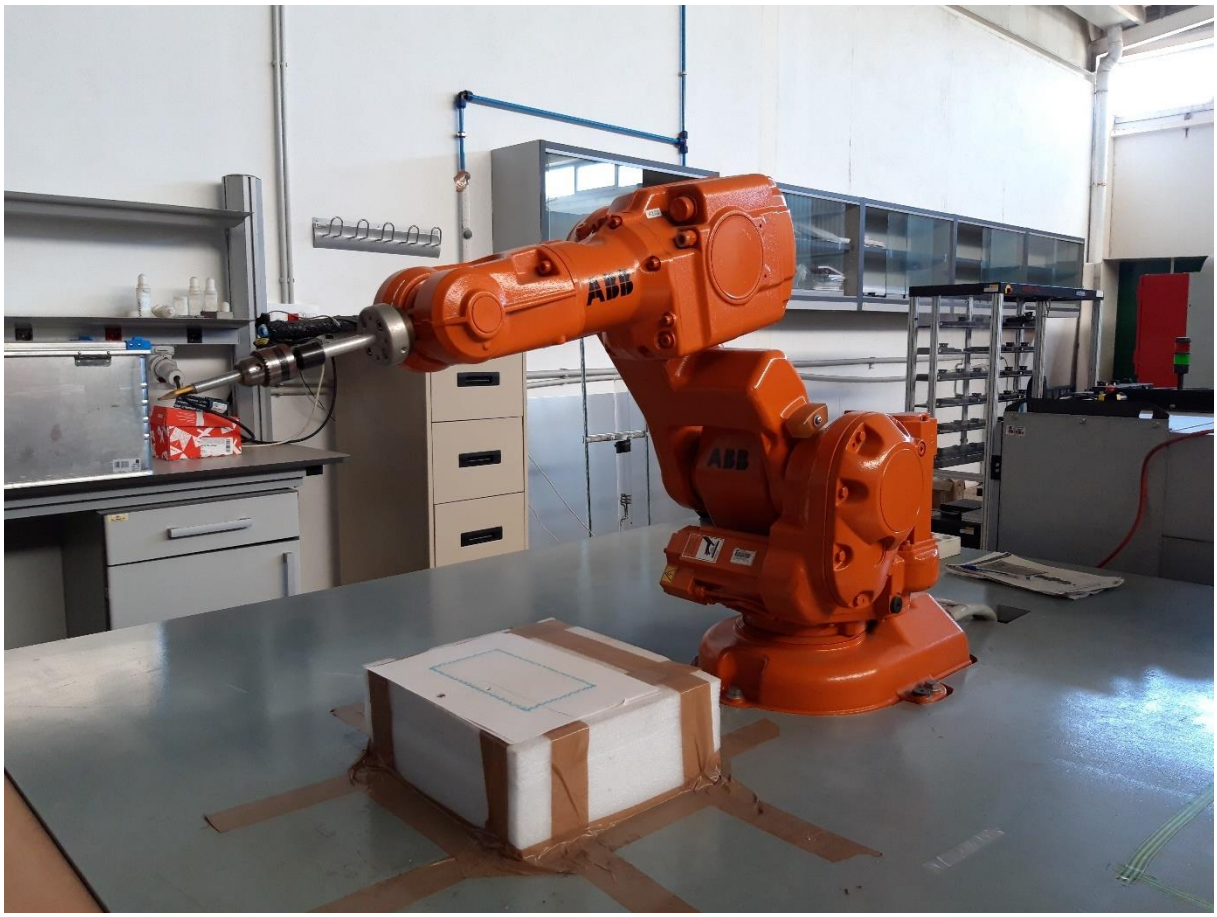



*Figure 1-4. Photograph of the ABB IRB 140 at the CITE IV building of the University of Almería.*

## 1.4 Interest

As it was seen in the brief summary of the history of robotics, it is getting increasingly common to see robots performing many different types of operations, with special interest in the industrial ones. These operations are diverse, from large assembly lines to small work cells.

The demands for highly qualified technicians and engineers with a good base knowledge in robotics are in consequence on the rise all around the globe. This master's thesis is desired to be as well, an appealing cover letter for a possible new job in the field of robotics.

## 1.5 Objectives

The main objective of this master's thesis is to delve into this broad field of robotics with the elaboration of a kinematic and dynamic model of the ABB IRB 140 robot based on a complete multidisciplinary analysis. Implemented in a feedback control schematic, these models will be then subjected to setpoints in form of predefined trajectories in order to obtain an optimum control of the mechatronic system as a whole.

During this process, a physical model will be obtained on MATLAB and on Simulink, a graphical programming environment integrated on MATLAB itself. The Simulink's SimScape Multibody library, which provides a multibody simulation environment for 3D mechanical systems, will be used to build the dynamic model of the robot.

Finally, the behavior of this model under a controller's action will be compared with that of the ABB Software RobotStudio to validate the whole schematic. The controller must be robust and with customizable aggressiveness.

To achieve all this, a list of secondary objectives was proposed:

1. **Kinematic model.** Generation of a simplified geometric model equivalent to that of the robot ABB IRB 140, which will be subject to several tests in order to validate this equivalence, consisting of:
    a. The introduction of joint angles as input to obtain the coordinates and orientation of the end effector, also called direct kinematics.
    b. The introduction of end coordinates and orientation as input to obtain the joint angles of the robot as output, or inverse kinematics.
2. **Dynamic model.** Introduction of other physical properties of the robot, such as a simplified 3D geometry, centers of mass, joint inertia matrices and material density. Calculation of the dynamic behavior, which relates the movement of the robot with the forces involved in it. The obtention of the dynamic model is one of the most complex aspects of robotics, with still room for improvement and optimization. This model allows to relate mathematically:
    a. The introduction of forces and torques applied at the joints or at the end of the robot as inputs to obtain the joint coordinates and their temporal evolution (angles, angular velocities and angular accelerations), or direct dynamics.

   b. The introduction of joint coordinates and their derivatives as inputs to obtain the forces and torques that should be exerted, or inverse dynamics. This is the dynamic model that will be implemented on the control system.

3. **Construction of a feedback control system**. This schematic will consist of:

   a. A RobotStudio data/trajectory generator block that generates in continuous time a trajectory of end effector poses. Every pose consists of 3 Cartesian coordinates plus a rotation quaternion, all relative to the coordinate origin of the robot. The block will calculate the first and second derivative of the poses too (velocity and acceleration).

   b. An inverse kinematics block. Built to transform the current pose plus its derivatives into joint coordinates and its derivatives.

   c. A controller block. The controller will obtain the appropriate torques for every joint needed to achieve the previously calculated required angular coordinates. In the case of CTC controllers, the symbolic dynamic model will be placed here.

   d. An ABB IRB 140 model block. Every output of the controller will be introduced in its correspondent robot joint, giving rise to certain joint coordinates of the robot plus derivatives.

   e. A direct kinematics block. These current joint coordinates pass through the direct kinematics block, which transforms them back into poses and their derivatives, this time of the real robot.

   f. A comparator block. The current end effector pose is feedbacked to the controller and at the same time compared with the initial desired pose at the beginning of the schematic. This difference determines the effectiveness of the complete control system in reaching the input setpoints.

4. **Model validation.** The comparator block errors of all developed control schematics will be compared with each other and the setpoints to evaluate their performance.

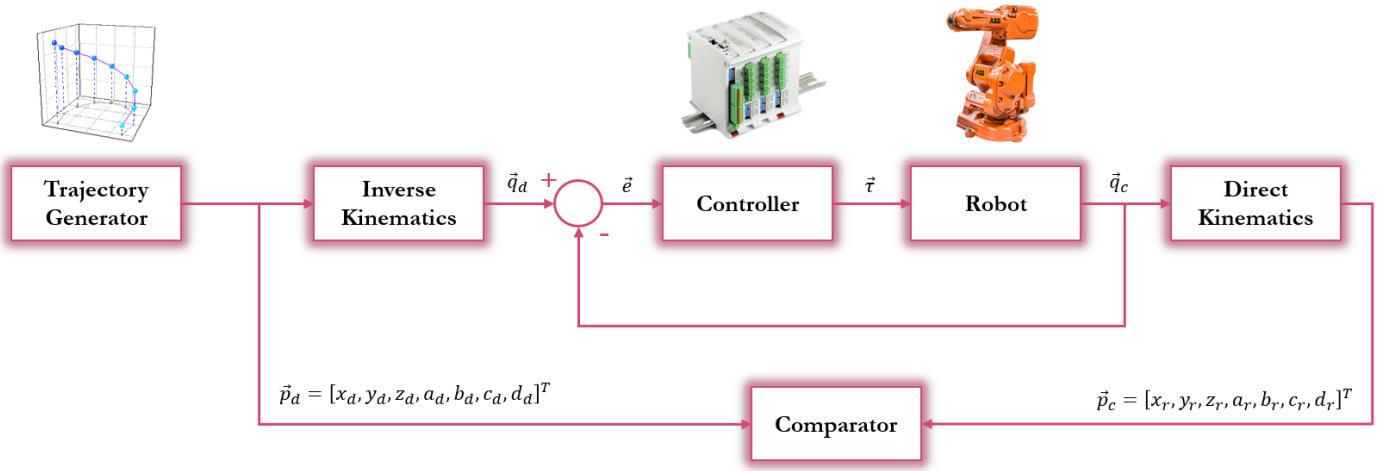



*Figure 1-5. Desired general schematic of the robot control model.*

## 1.6 Stages of development

The master's thesis goes through the following stages:

1. **Knowledge acquisition for the adequate understanding of the kinematics and dynamics of a robot arm,** its methodology, formulas, and state of the art of robotics. Study of control algorithms and characteristics of the ABB IRB 140 model.
2. **Software download**. A list with the necessary software will be established, containing the programs for coding and simulating robot arm features.
3. **Main schematic drafting of the robot system**. A schematic that gathers the whole control loop will be drafted.
4. **Kinematic modeling.** Two different functions will be programmed in MATLAB for each of the kinematic models, direct and inverse, which only depends on the joint types and distances between them.
5. **Dynamic modeling.** Using CAD models of the robot arm downloaded from the ABB website, SimScape blocks will be able to calculate the centers of mass and moments and products of inertia of the robot, based on spatial properties. These blocks will constitute the robot system on which tests will be performed. Furthermore, based upon the data obtained by the SimScape blocks, a dynamic model of the robot will be generated through the Newton-Euler algorithm.
6. **Development of an accurate and robust controller** that can handle any robot configuration requirement. Either by means of a PID controller or a Computed Torque Controller, the robot must be able to follow certain trajectories in the times set for it with the least possible error.
7. **Equivalence tests between the models' behavior and RobotStudio's output**. The results of the model will be compared with RobotStudio's output to evaluate the validity of the models and apply corrective actions if necessary.
8. **Analysis of results and conclusions** to evaluate how close the control system got to RobotStudio's results and what can be done to improve it.

## 1.7 Bibliographic review

Although there is some bibliography available focused on the ABB IRB 140, it is not as thorough and detailed as it would be desirable. Specially regarding the dynamics of the robot. Few works develop the details of the obtention of the dynamic model and none of them apply Computed Torque Controllers as control solution for trajectory tracking. For this reason, the implementation of a Computed Torque Controller on an ABB IRB 140 model would be completely novel.

The most basic features about the ABB IRB 140 can be found in the Product Specification [1]. Many characteristics are not included due to confidentiality reasons, characteristics that would be very useful in this master's thesis, but they will have to be extrapolated or simplified. Therefore, the product specification will provide mainly with the spatial measures and limitations of the robot.

The desired method to obtain the kinematic model of the robot is the Denavit-Hartenberg algorithm. This was developed back in 1955 by Jacques Denavit and Richard Hartenberg and it can be found in [2]. An application of the algorithm on the ABB IRB 140 is presented in [3], [5] and [6]. [3] proposes a geometrical inverse kinematic solution for the first three links. On the other hand, Suárez Baquero, M. & Ramírez Heredia [4] use the Screw Successive Displacements method for the direct kinematics and geometric strategies for the inverse kinematics.

Other works are more focused on practical application of robot, such as Córdoba López [5], who develops the Denavit-Hartenberg method as a theoretical introduction for further programming on RobotStudio, or Mato San José [6] who simulates the kinematics and dynamics of the robot implementing these using the MATLAB's Robotic Toolbox.

Rønnestad [7] begins with the parameter identification for the dynamic model of the 6th robot joint by using the Least Squares (LS) estimation method, followed by the development of a controller for this dynamic model. It includes the possibility to handle different tools at the end effector.

The most used bibliographic resource was [8], a robotics book by Barrientos, which covers those issues related to the operation of a robot: mathematical, mechanical and control aspects. Robot kinematics and dynamics are broadly detailed here and have entailed the main source of knowledge for this master's thesis. Followed by Corke's [9] book, deeply thorough as well, which contains the theoretical basis for the Computed Torque Controller, as well as more mathematical approaches of robot kinematics and dynamics. Kelly, R, and Santibáñez, V. with their book [10] introduce abundant control solutions for robot systems and help to broaden the theoretical basis of the Computed Torque Controllers.

Sarabandi, S., & Thomas, F. [11] develop in their paper a mathematical algorithm to compute numerically a quaternion from a rotation matrix without falling in ill-conditioned situations that may compromise the stability of a solution due to rounding issues.

Finally, [12] and [13] serve as examples of works with parameter identification for future improvements and expansions of the master's thesis.

## 1.8 Timeline

For the achievement of all the proposed objectives, a timeline was developed. Each task corresponds to a stage of the development. This planning was distributed in fortnights as shown in Table 1-1.

| Month | March | | April | | May | | June | | July | |
|---|---|---|---|---|---|---|---|---|---|---|
| Fortnight / Task | 1ª | 2ª | 1ª | 2ª | 1ª | 2ª | 1ª | 2ª | 1ª | 2ª |
| Knowledge acquisition | | | | | | | | | | |
| Software downloads | | | | | | | | | | |
| Software design and pseudocode | | | | | | | | | | |
| Kinematic model | | | | | | | | | | |
| Dynamic model | | | | | | | | | | |
| Controller development | | | | | | | | | | |
| Equivalence tests | | | | | | | | | | |
| Comparisons and conclussion | | | | | | | | | | |

*Table 1-1. Timeline of the stages of development.*

# Chapter 2. Material and methods

The following chapter presents the material needed to build the model, both software and hardware and the mathematical models.

## 2.1 Material

### Computer with Windows 7

Microsoft Windows 7 is the oldest version of the Windows OS that still has an update service. It has been chosen as the operating system on which to run the software for its ease of use and presence in most laboratories at the University of Almería.

In addition, any Simulink model and MATLAB code can be executed in any operative system.

### MATLAB 2019a

MATLAB is numerical computing environment that allows to program code making use of its multiple integrated functions and libraries, which facilitate the characterization of complex mathematical models.

Integrated as well in MATLAB, there is Simulink, a graphical programming environment for modeling, simulating and analyzing multidomain dynamical systems. The control loop model will be fully developed in this tool with the aid of libraries such as the SimScape Multibody library.

### SolidWorks

SolidWorks is a CAD (Computer Aided Design) and CAE (Computer Aided Engineering) software for Windows. It allows to represent in 3D all links of the robot in order to load them on Simulink and extract their physical properties.

### ABB RobotStudio

RobotStudio is a software for simulation and offline programming designed by ABB that allows to replicate real workspaces with numerous available ABB robots and tools. In these virtual workspaces it is possible to simulate desired robot behaviors without shutting down production in plant (Figure 3-1).

This makes RobotStudio a perfect application for training and learning. In a certain sense this master's thesis seeks to create a simulation environment too, that represents the robot physics accurately.

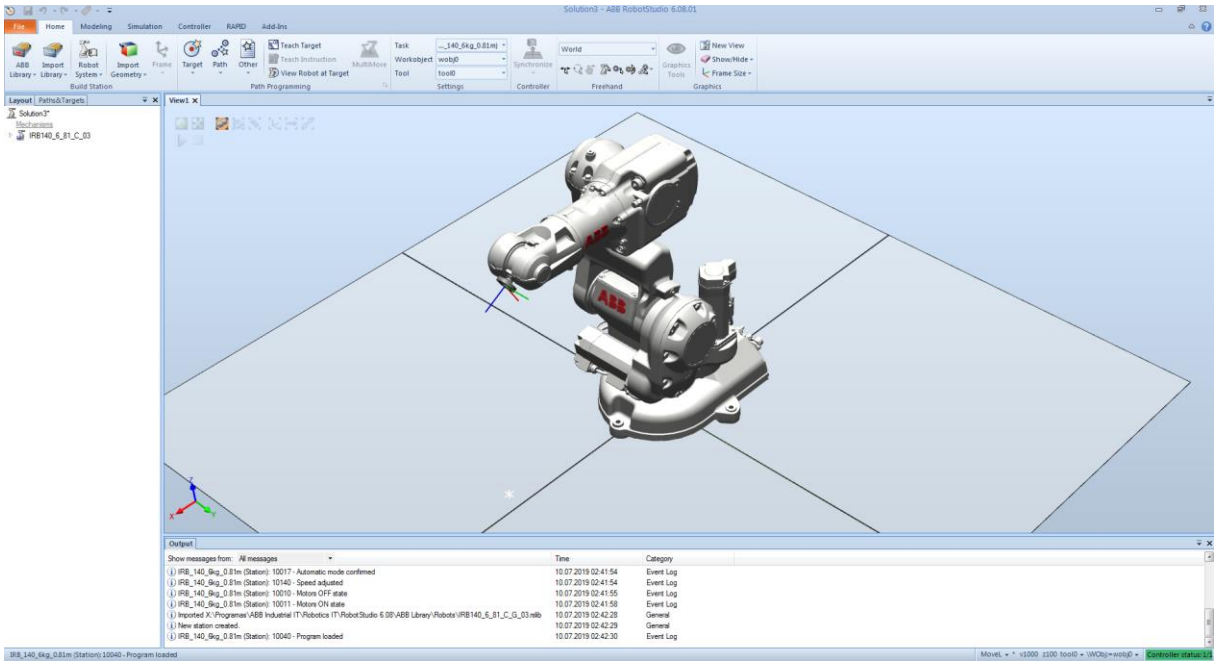


*Figure 2-1. Start screen for a solution workspace consistent of an ABB IRB 140.*

## 2.2 Methods

### 2.2.1 Robot kinematics

Kinematics is the branch of mechanics that studies the motion of a solid or system of bodies with respect to a reference system without considering any forces or torques that may intervene. In this manner, robot kinematics only cares about the analytical description of movement as a function of time and particularly about the relation between the pose (location and orientation) of the end effector and joint angles.

A robot arm, also known as a serial-link manipulator, consists of solid rigid links and joints. Joints can be rotational or translational, and their motion modifies the relative pose of the

subsequent links. The beginning of the link chain is usually fixed, and the end is free to move in space.

There are two fundamental problems in robot kinematics, one is **direct kinematics**, whose purpose is to determine and map the pose of the robot end effector as function of its joint angles. The other is **inverse kinematics**, which is the opposite. Starting off with a desired pose of the end effector, finds a suitable joint configuration in order to reach that pose.

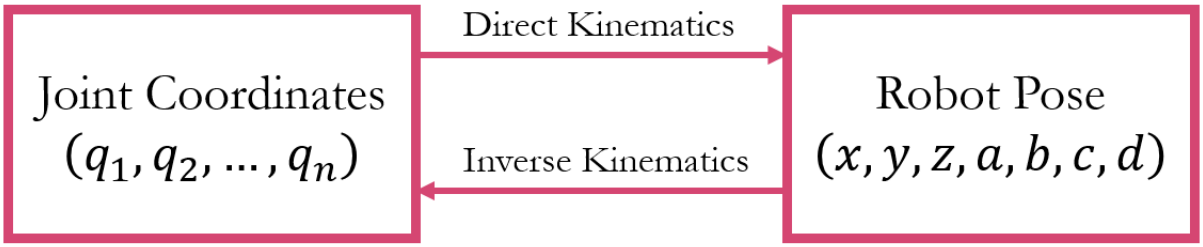



*Figure 2-2. Diagram of the relationship between direct and inverse kinematics.*

Kinematics is also capable of finding the relation between a differential change in the velocities and accelerations of the joints and a differential change of the velocity and acceleration of the end effector and the rate of change of the rotation quaternion and its derivative.

This differential model is defined by the **Jacobian matrix**. It relates the vector of joint velocities with the vector of linear and rotational velocities, as another section of the direct kinematic problem focused on velocity.

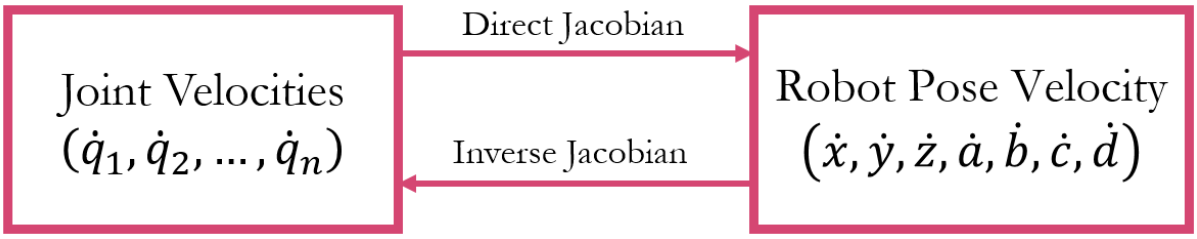



*Figure 2-3. Diagram of the relationship between direct and inverse Jacobian matrices.*

#### *2.2.1.1 The direct kinematic problem*

Jacques Denavit and Richard Hartenberg proposed a systematic method in 1955 to describe and represent the spatial geometry of the elements of a kinematic chain, and in particular of a robot, with respect to a fixed reference system [2]. This method uses a homogenous transformation matrix to describe the spatial relationship between two adjacent rigid elements, reducing the direct kinematic problem to find a homogenous transformation matrix $4 \times 4$ that relates the spatial location of the robot end effector with respect to the coordinate system of its base.

Choosing the coordinate systems associated with each link according to the representation proposed by Denavit-Hartenberg (from now on, D-H), it is possible to jump from

one to the next by means of 4 basic transformations that depend exclusively on the geometric characteristics of the link.

It should be noted that although in general a homogeneous transformation matrix is defined by 6 degrees of freedom (three linear translations and three angular rotations), the D-H method allows, in rigid links, to reduce this number to 4 with a correct choice of coordinate systems. These 4 basic transformations consist of a succession of rotations and translations that allow to relate the reference system of the element $i-1$ with the system of the element $i$. The transformations in question are as follows:

1. **Rotation** around the $z_{i-1}$ axis of an angle $\theta_i$.
2. **Translation** along $z_{i-1}$ a distance $d_i$; vector $d_i$ $(0,0,d_i)$.
3. **Translation** along $x_i$ a distance $a_i$; vector $a_i$ $(a_i,0,0)$.
4. **Rotation** around the $x$-axis an angle $\alpha_i$.

Since the product of matrices is not commutative, the transformations must be carried out in the indicated order. Making use of homogeneous transformation matrices, every transformation can be represented as:

$$^{i-1}A_i = Rot(\theta_i)\cdot Tra(0,0,d_i)\cdot Tra(a_i,0,0)\cdot Rot(\alpha_i) \tag{2.1}$$

$$^{i-1}A_i = \begin{bmatrix} c(\theta_i) & -s(\theta_i) & 0 & 0 \\ s(\theta_i) & c(\theta_i) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c(\alpha_i) & -s(\alpha_i) & 0 \\ 0 & s(\alpha_i) & c(\alpha_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

$s(x)$ y $c(x)$ are the functions sine and cosine respectively. Multiplying, the generic form of a D-H transformation matrix is:

$$^{i-1}A_i = \begin{bmatrix} c(\theta_i) & -c(\alpha_i)s(\theta_i) & s(\alpha_i)s(\theta_i) & a_i c(\theta_i) \\ s(\theta_i) & c(\alpha_i)c(\theta_i) & -s(\alpha_i)c(\theta_i) & a_i s(\theta_i) \\ 0 & s(\alpha_i) & c(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

With a total of $n$ joints, the transformation matrix from the origin of the system $\{S_0\}$ to its end $\{S_n\}$ is:

$$T = {}^0A_1 \cdot {}^1A_2 \cdot \ldots \cdot {}^{n-1}A_n \tag{2.4}$$

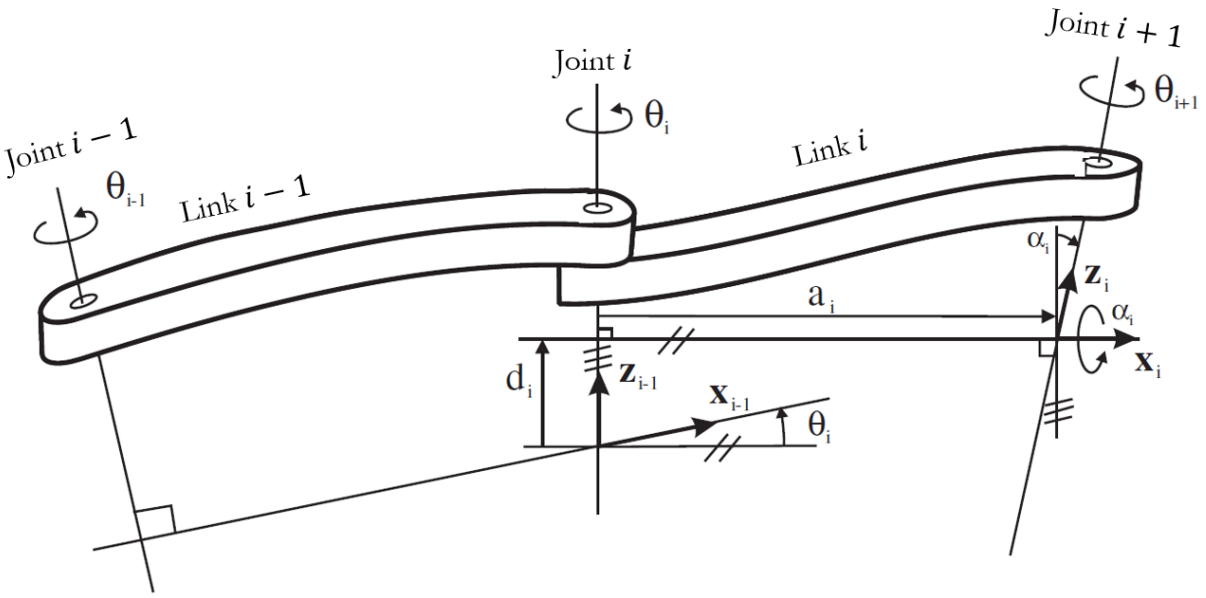



*Figure 2-4. D-H parameters for a rotational joint.*

In order to achieve this simplified configuration, certain rules must be established. These, in addition to the definition of the 4 D-H parameters, make up the following Denavit-Hartenberg algorithm for the resolution of the direct kinematic problem:

**D-H 1.** Label the robot links starting from 1 (first free moving robot link) and ending at $n$ (last free moving robot link). The fixed base of the robot will be labeled as 0.

**D-H 2.** Label each joint starting from 1 (first degree of freedom) and ending at $n$.

**D-H 3.** Pinpoint the axis of every joint. If it is rotational, the axis will be its own rotating axis. If it is prismatic it will be the axis along its displacement occurs.

**D-H 4.** For $i$, from 0 to $n-1$, place the axis $z_i$ where the joint axis $i+1$ stands.

**D-H 5.** Place the origin of the coordinate system $\{S_0\}$ at any point of the axis $z_0$. The axis $x_0$ and $y_0$ will be positioned so that they form a dextrorotatory system with $z_0$.

**D-H 6.** For $i$, from 1 to $n-1$, place the origin of the coordinate system $\{S_i\}$ (attached to link $i$) at the intersection of the axis $z_i$ with the shared perpendicular line of $z_{i-1}$ and $z_i$. If both axes intersected, $\{S_i\}$ would be placed at that intersecting point. If they were parallel $\{S_i\}$ would be place at the joint $i+1$.

**D-H 7.** Place $x_i$ at the shared perpendicular line of $z_{i-1}$ and $z_i$.

**D-H 8.** Place $y_i$ so that it draws a dextrorotatory system with $x_i$ and $z_i$.

**D-H 9.** Place the coordinate system $\{S_i\}$ at the end effector of the robot, so that $z_n$ matches with the direction of $z_{n-1}$ and $x_n$ is perpendicular to $z_{n-1}$ and $z_n$.

**D-H 10.** Obtain $\theta_i$ as the angle to be rotated around $z_{i-1}$, so that $x_{i-1}$ and $x_i$ are parallel.

**D-H 11.** Obtain $d_i$ as the distance measured along $z_{i-1}$, that $\{S_{i-1}\}$ would have to be shifted, so that $x_i$ and $x_{i-1}$ are aligned.

**D-H 12.** Obtain $a_i$ as the distance measured along $x_i$ (which would now coincide with $x_{i-1}$) that $\{S_{i-1}\}$ would have to be shifted so that its origin coincides with $\{S_i\}$.

**D-H 13.** Obtain $\alpha_i$ as the angle to be rotated around $x_i$, so that the new $\{S_{i-1}\}$ would coincides with $\{S_i\}$

**D-H 14.** Obtain all transformation matrices ${}^{i-1}A_i$.

**D-H 15.** Obtain the transformation matrix that relates the base coordinate system $\{S_0\}$ with the end effector coordinate system $\{S_6\}$: $T = {}^{0}A_1 \cdot {}^{1}A_2 \cdot \ldots \cdot {}^{n-1}A_n$.

**D-H 16.** This transformation matrix $T$ generates a rotation matrix and a translation vector as function of every joint coordinate.

A homogeneous transformation matrix $T$ is a $4 \times 4$ matrix that represents the transformation of a vector of homogeneous coordinates from one coordinate system to another.

A homogenous transformation matrix has the generic form:

$$T = \begin{bmatrix} R_{3\times3} & t_{3\times1} \\ f_{1\times3} & w_{1\times1} \end{bmatrix} = \begin{bmatrix} Rotation & Translation \\ Perspective & Scale \end{bmatrix} \tag{2.5}$$

In robotics, perspective and scaling components are assumed to be zero and one respectively, being only relevant the rotation and translation sub-matrices.

The **translation matrix** is the most straightforward, since it allows the extraction of the translation vector instantaneously. This vector is a column vector whose 3 values represent the translation of the coordinate system on the $x$, $y$ and $z$ axes of the original system.

$$[t_{3\times1}] = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2.6}$$

On the other hand, the **rotation matrix** requires some processing of its values to extract useful information about the rotation to which the coordinate system has been subjected. It does not therefore have a trivial solution. In this master's thesis, the representation of the rotation between two coordinate systems by means of quaternions has been opted.

A **quaternion** is a system of representation of rotations consisting of a vector of four components, one real and three complex. It is a tool of great versatility. Although they are not as intuitive as Euler's angles, quaternions are easy to calculate, they are efficient, they do not have singularities and they do not suffer from ambiguity.

$$q = [a, b, c, d]^T \tag{2.7}$$

The selected algorithm to calculate them has been the Sarabandi-Thomas method. It can be found in [9]. Code 1 and Code 2 implement it numerically and symbolically respectively. Its mathematical formulas are the following:

$$a = \begin{cases} \frac{1}{2} \cdot \sqrt{1 + R_{11} + R_{22} + R_{33}} & if\ R_{11} + R_{22} + R_{33} > \eta \\ \frac{1}{2} \cdot \sqrt{\frac{(R_{32} - R_{23})^2 + (R_{13} - R_{31})^2 + (R_{21} - R_{12})^2}{3 - R_{11} - R_{22} - R_{33}}} & otherwise \end{cases} \quad (2.8)$$

$$b = \begin{cases} \frac{1}{2} \cdot \sqrt{1 + R_{11} - R_{22} - R_{33}} & if\ R_{11} - R_{22} - R_{33} > \eta \\ \frac{1}{2} \cdot \sqrt{\frac{(R_{32} - R_{23})^2 + (R_{12} + R_{21})^2 + (R_{31} + R_{13})^2}{3 - R_{11} + R_{22} + R_{33}}} & otherwise \end{cases} \quad (2.9)$$

$$c = \begin{cases} \frac{1}{2} \cdot \sqrt{1 - R_{11} + R_{22} - R_{33}} & if\ -R_{11} + R_{22} - R_{33} > \eta \\ \frac{1}{2} \cdot \sqrt{\frac{(R_{13} - R_{31})^2 + (R_{12} + R_{21})^2 + (R_{23} + R_{32})^2}{3 + R_{11} - R_{22} + R_{33}}} & otherwise \end{cases} \quad (2.10)$$

$$d = \begin{cases} \frac{1}{2} \cdot \sqrt{1 - R_{11} - R_{22} + R_{33}} & if\ -R_{11} - R_{22} + R_{33} > \eta \\ \frac{1}{2} \cdot \sqrt{\frac{(R_{21} - R_{12})^2 + (R_{31} + R_{13})^2 + (R_{32} + R_{23})^2}{3 + R_{11} + R_{22} - R_{33}}} & otherwise \end{cases} \quad (2.11)$$

Where $R$ is the rotation matrix and $\eta$ the selected threshold ($= 0$ for best performance). The direct kinematic problem for position and rotation would be now solved, and the output pose vector, combining the translation vector and the quaternion would be:

$$p = \begin{bmatrix} x \\ y \\ z \\ a \\ b \\ c \\ d \end{bmatrix} \quad (2.12)$$

### Jacobian Matrix

The version of this problem applied for velocities must make use of the Jacobian matrix, as said in the introduction of section 2.2.1 In the case of a robot with $n$ joints:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{a} \\ \dot{b} \\ \dot{c} \\ \dot{d} \end{bmatrix} = J \cdot \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \vdots \\ \dot{q}_n \end{bmatrix} \quad (2.13)$$

Where $J$ is the Jacobian matrix:

$$J = \underbrace{\begin{bmatrix} \frac{\partial f_x}{\partial q_1} & \cdots & \frac{\partial f_x}{\partial q_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_d}{\partial q_1} & \cdots & \frac{\partial f_d}{\partial q_n} \end{bmatrix}}_{7\times n} \tag{2.14}$$

And $f_x, f_y, \dots, f_d$ are all the direct kinematic formulas that relate every pose parameter as a function of all joint angles.

$$\begin{matrix} x = f_x(q_1, \dots, q_n) & y = f_y(q_1, \dots, q_n) & z = f_z(q_1, \dots, q_n) \\ a = f_a(q_1, \dots, q_n) & b = f_b(q_1, \dots, q_n) & c = f_c(q_1, \dots, q_n) \quad d = f_d(q_1, \dots, q_n) \end{matrix} \tag{2.15}$$

As can be observed, every joint configuration generates a different Jacobian matrix. In the case that the rank of the matrix reaches a number lower than the total of degrees of freedom, it means that one or more columns are equal to a linear combination of other columns. The robot is in a singularity, where one or more degrees of freedom are lost. The physical meaning of this phenomenon is that any effort to vary some pose parameter of the end effector by the movement of a certain joint would not be possible.

The pose accelerations can be obtained mathematically processing the Jacobian matrix formula, which is a differential equation.

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \\ \ddot{a} \\ \ddot{b} \\ \ddot{c} \\ \ddot{d} \end{bmatrix} = J \cdot \begin{bmatrix} \ddot{q}_1 \\ \ddot{q}_2 \\ \vdots \\ \ddot{q}_n \end{bmatrix} - \frac{dJ}{dt} \cdot \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \vdots \\ \dot{q}_n \end{bmatrix} \tag{2.16}$$

#### *2.2.1.2 The inverse kinematic problem*

One way of obtaining the inverse kinematic problem is through finding geometrical relationships between links and joints of the robots, but it is inadequate due to the complexity of the ABB IRB 140. With 6 degrees of freedom it would take too much time to find an analytical solution. Therefore, another method is preferred.

This other method would be the resolution by means of the homogeneous transformation matrix $T$. It would be starting from 12 nonlinear equations present in the matrix, 6 of them linearly dependent. Through a numerical method, all joint angles necessary to reach a certain pose would be calculated, making it a feasible task due to the nature of the simulation. However, it is important to note that there might be multiple solutions for a single objective pose, or even no solution at all.

A good graphic example of this feature are the 2 robot configurations below (Figure 2-5) simulated on RobotStudio, where they reach the same pose out of different joint angles.

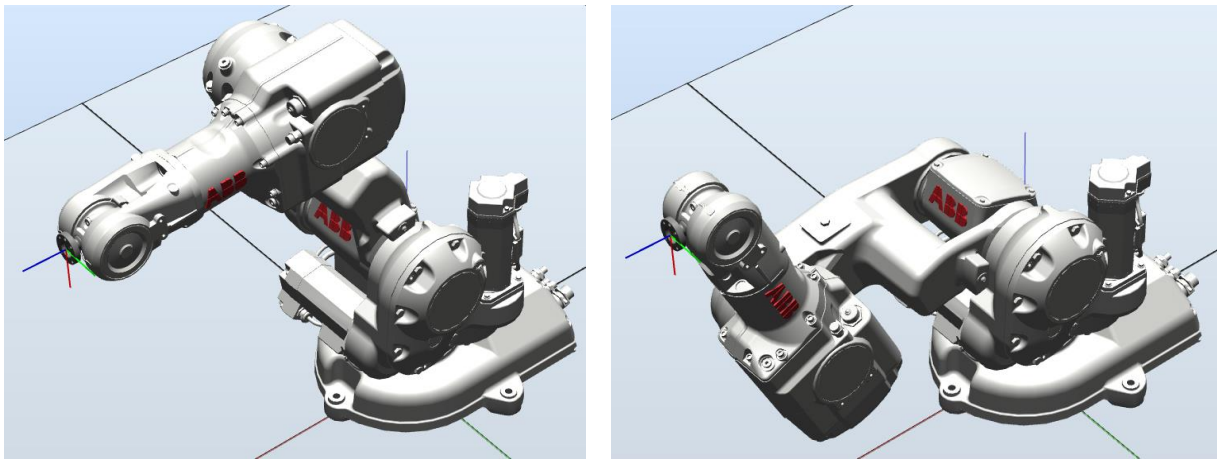


*Figure 2-5. Example of 2 different configurations that end up at the same end effector pose.*

On the other hand, a non-existence of a solution appears when the requested pose is located outside the work area, spatial-wise or orientation-wise.

Returning to the Jacobian matrix, if it relates the pose velocity with respect to the joint angles and joint velocities, its inverse should obtain the joint velocities out of joint angles and the pose velocity.

$$\begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \vdots \\ \dot{q}_n \end{bmatrix} = J^{-1} \cdot \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{a} \\ \dot{b} \\ \dot{c} \\ \dot{d} \end{bmatrix} \tag{2.17}$$

Unfortunately, two problems emerge. The first one is, if the rotation representation chosen are quaternions, the number of pose parameters would be 7 (3 spatial and 4 rotational), $J$ would not be square and therefore, its inverse could not be worked out. The second is that such large symbolic matrices as $J$ if it was square, are not computable in reasonable intervals of time.

The most computationally efficient method would be to solve the linear system for the current joint configuration, where if $Ax = b$ represents the standard matrix form of the linear system:

$$Ax = b \rightarrow \underbrace{J}_{A} \cdot \underbrace{\begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \vdots \\ \dot{q}_n \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{a} \\ \dot{b} \\ \dot{c} \\ \dot{d} \end{bmatrix}}_{b} \tag{2.18}$$

And finally, the obtention of the joint accelerations as function of the pose, pose velocity and pose acceleration, differentiating formula 2.18 with respect to time and reordering it to fit the standard matrix form:

$$\underbrace{J}_{A} \cdot \underbrace{\begin{bmatrix} \ddot{q}_1 \\ \ddot{q}_2 \\ \vdots \\ \ddot{q}_n \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \\ \ddot{a} \\ \ddot{b} \\ \ddot{c} \\ \ddot{d} \end{bmatrix} + \frac{dJ}{dt} \cdot \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \vdots \\ \dot{q}_n \end{bmatrix}}_{b} \tag{2.19}$$

### 2.2.2 Robot dynamics

Robot dynamics addresses the relation between the forces acting on a solid or group of solids and the movement that originates from them. This relation is calculated through the so-called dynamic model, which establishes the mathematical association between:

- The location of the robot defined by its joint variables or by the pose of its end effector, and its derivatives: velocity and acceleration.
- The forces and torques applied in the joints (or at the end effector).
- The dimensional parameters of the robot, such as length, mass and inertia of its links.

The process of obtaining this model for mechanisms of one or two degrees of freedom is not excessively complex, but as the number of degrees of freedom increases, the computation of the dynamic model becomes exponentially more complicated. For this very reason, most of the time the best option is to implement iterative numerical methods.

The process of obtaining the dynamic model has been proven one of the most complex aspects of robotics, which has led to be obviated on numerous occasions. However, the dynamic model is essential to achieve the following purposes:

- Simulation of the movement of the robot.
- Design and evaluation of the mechanical structure of the robot.
- Dimensioning of the actuators.
- Design and evaluation of the dynamic control of the robot.

This last purpose is of great importance, since the precision of positioning and velocity depends vastly on the quality of the dynamic model developed.

A perfect dynamic model would consider many other elements of the robots aside from links and gravity itself. Actuators, internal and external temperature or wear among others, add up to some inequalities between the model and reality that may impact the quality of control. These will be ignored in this master's thesis for the sake of simplicity.

#### *2.2.2.1 The rigid-body equations of motion*

For a set of links, their rigid-body equations of motion can be represented elegantly as a matrix set of differential equations:

$$Q = M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) + Others \tag{2.20}$$

Where $Q$ ys the vector of generalized actuator forces associated with the generalized coordinates $q$, $M$ is the joint-space inertia matrix, $C$ is the Coriolis and centripetal coupling matrix, and $G$ is the gravity loading. Others represent the ignored effects that have a smaller impact on the robot control performance.

#### *2.2.2.2 The Lagrange method*

Obtaining the dynamic model of a robot from the Lagrangian formulation leads to an algorithm with a computational cost of order $O(n^4)$. That means, the number of operations to be carried out increases with the fourth power of the number of degrees of freedom of the robot. In the particular case of this thesis, with a robot with 6 degrees of freedom, this number of operations makes the algorithm presented rather unsuitable to be used in real time.

#### *2.2.2.3 The Newton-Euler method*

This new algorithm is based on vector operations (with scalar and vector products between vector magnitudes, and products of matrices and vectors). This makes it far more efficient than the Lagrangian formulation. In fact, the order of computational complexity of the Newton-Euler recursive formulation is $O(n)$ which indicates that it depends proportionally on the number of degrees of freedom.

For a robot with only rotational joints, the method starts from the torque equilibrium formula:

$$\sum \tau = I \cdot \omega + \omega \times (I \cdot \omega) \tag{2.21}$$

Through an adequate processing of this equation, the Newton-Euler method recursively formulates the equilibrium equations of forces and torques (only torques are needed for this particular robot), so that the position, velocity and acceleration of the link $i$ referred to the base of the robot are obtained from those corresponding to the link $i-1$ and the relative movement of the joint $i$. Initiating the method from link 1, it stops at link $n$.

Counting with these position, speed and acceleration data, the forces and torques that act on the links are calculated, from link $n$ to the link $1$.

The step-by-step procedure for an only rotational joint robot is as follows:

**N-E 1.** Assign to the $n$ links a coordinate system in accordance with the D-H algorithm.

**N-E 2.** Establish initial conditions:

For the coordinate system of the robot base $\{S_0\}$ supposing it is not moving:

- Zero angular velocity, ${}^0\omega_0 = [0 \quad 0 \quad 0]^T$
- Zero angular acceleration, ${}^0\dot{\omega}_0 = [0 \quad 0 \quad 0]^T$
- Zero linear velocity, ${}^0v_0 = [0 \quad 0 \quad 0]^T$
- Linear acceleration in terms of gravity, ${}^0v_0 = [0 \quad 0 \quad -g]^T$
- Auxiliary vector $z_0 = [0 \quad 0 \quad 1]^T$
- Coordinates of the origin of the system $\{S_i\}$ with regard to $\{S_{i-1}\}$ ${}^ip_i = [a_i \quad d_i s(\alpha_i) \quad d_i c(\alpha_i)]$

- Coordinate of the center of mass of the link $i$ with regard to $\{S_i\}$: ${}^{i}s_i$
- Inertia matrix of the link $i$ with regard to its center of mass expressed on $\{S_i\}$: ${}^{i}I_i$

**N-E 3.** Obtain the rotation matrices ${}^{i-1}R_i$ and their inverses ${}^{i}R_{i-1} = \left({}^{i-1}R_i\right)^{-1} = \left({}^{i-1}R_i\right)^{T}$

**N-E 4.** *(For $i$, from $1$ to $n$, perform steps N-E 4 to N-E 7)* Obtain the angular velocity of the system $\{S_i\}$.

$$ {}^{i}\omega_i = {}^{i}R_{i-1}\left({}^{i-1}\omega_{i-1} + z_0\dot{q}_i\right) \tag{2.22}$$

**N-E 5.** Obtain the angular acceleration of the system $\{S_i\}$.

$$ {}^{i}\dot{\omega}_i = {}^{i}R_{i-1}\left({}^{i-1}\dot{\omega}_{i-1} + z_0\ddot{q}_i\right) + {}^{i-1}\omega_{i-1} \times z_0\dot{q}_i \tag{2.23}$$

**N-E 6.** Obtain the linear acceleration of the system $\{S_i\}$.

$$ {}^{i}\dot{v}_i = {}^{i}\dot{\omega}_i \times {}^{i}p_i + {}^{i}\omega_i \times \left({}^{i}\omega_i \times {}^{i}p_i\right) + {}^{i}R_{i-1}\,{}^{i-1}\dot{v}_{i-1} \tag{2.24}$$

**N-E 7.** Obtain the linear acceleration of the center of mass of the link $i$.

$$ {}^{i}a_i = {}^{i}\dot{\omega}_i \times {}^{i}s_i + {}^{i}\omega_i \times \left({}^{i}\omega_i \times {}^{i}s_i\right) + {}^{i}\dot{v}_i $$

**N-E 8.** *(For $i$, from $n$ to $1$, perform steps N-E 8 to N-E 10)* Obtain the force exerted on the link $i$.

$$ {}^{i}f_i = {}^{i}R_{i+1}\,{}^{i+1}f_{i+1} + m_i\,{}^{i}a_i $$

**N-E 9.** Obtain the torque exerted on the link $i$.

$$ \begin{aligned} {}^{i}n_i = {}^{i}R_{i+1}\left[{}^{i+1}n_{i+1} + \left({}^{i+1}R_i\,{}^{i}p_i\right) \times {}^{i+1}f_{i+1}\right] + \\ \left({}^{i}p_i + {}^{i}s_i\right) \times m_i\,{}^{i}a_i + {}^{i}I_i\,{}^{i}\omega_i + {}^{i}\omega_i \times \left({}^{i}I_i\,{}^{i}\omega_i\right) \end{aligned} \tag{2.25}$$

**N-E 10.** Obtain the torque exerted on the joint $i$.

$$ {}^{i}\tau_i = {}^{i}n_i^{T}\,{}^{i}R_{i-1}z_0 \tag{2.26}$$

After these steps, the model would be finished. There would be a total of $n$ equations of torque, one for each joint, as function of joint variables: angles, velocities and accelerations.

### 2.2.3 The PID Controller

A PID Controller (Proportional-Integral-Derivative), or just PID, is a system used for building feedback control loops in multitude of applications that require continuously modulated control.

PIDs bases itself on the calculation of an error value $e(t)$, the difference between a desired setpoint and a current measured process value. This value is then processed separately three times. One of them is integrated, another one derived and all three multiplied by three different constants respectively. The mathematical structure of a parallel PID is:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)\, dt + K_d \frac{de(t)}{dt} \tag{2.27}$$

There are a couple more different structures, but in this master's thesis only the parallel one will be used.

### 2.2.4 The Computed Torque Controller

The dynamic model that characterizes the behavior of manipulative robots is generally nonlinear in terms of state variables (joint angles and joint velocities). This peculiarity of the dynamic model could suggest that given any controller, the differential equation that governs the closed-loop control system should also be non-linear in the corresponding state variables.

However, there is a non-linear controller in the state variables with which the closed-loop control system can now be described by means of a linear differential equation. Said controller can satisfy motion control objectives globally with a trivial selection of its design parameters. This is the Computed Torque Controller. Its principle is that the robot system dynamics and its inverse (the controller) are concatenated so that the overall system has a constant unity gain. Due to inconsistencies between the robot real behavior and the model a feedback loop is required to deal with errors.

The torque signal provided by the controller is:

$$Q = M(q^{\#})\left[\ddot{q}^{*} + K_v(\dot{q}^{*} - \dot{q}^{\#}) + K_p(q^{*} - q^{\#})\right] + C(q^{\#}, \dot{q}^{\#})\dot{q}^{\#} + G(q^{\#}) \tag{2.28}$$

Where $*$ denotes a desired value and $\#$ a current value. $K_p$ and $K_v$ are the positive-definite design matrices of position and velocity respectively.

Despite the presence of the term $K_v(\dot{q}^{*} - \dot{q}^{\#}) + K_p(q^{*} - q^{\#})$ in the control law, these are actually multiplied by the matrix of inertia $M$. This effect results in the fact that the control law has a term of the PD kind, but this is not a linear controller, since the position and velocity gains are not constant but depend explicitly on the position error $q^{*} - q^{\#}$.

For practical purposes, the design matrices $K_p$ and $K_v$ can be diagonal, therefore the closed-loop equation represents a decoupled multivariable linear system, that is, the dynamic behavior of joint angles is governed by second-order linear differential equations, where each of them is independent of the rest. In this context the choice of $K_p$ and $K_v$ matrices can be written specifically as:

$$\begin{aligned} K_p &= diag\{\omega_1^2, \dots, \omega_n^2\} \\ K_v &= diag\{2\omega_1, \dots, 2\omega_n\} \end{aligned} \tag{2.29}$$

With this choice, each joint responds the same as a critically damped linear second order system with bandwidth $\omega_i$. The bandwidth $\omega_i$ determines the response speed of the junction and, consequently, the exponential decay rate of the errors $\dot{q}^{*} - \dot{q}^{\#}$ and $q^{*} - q^{\#}$.

# Chapter 3. Implementation and results

In this chapter, the implementation of the theory and methods present in Chapter 2 will be detailed, exposed and explained, highlighting results.

## 3.1 Robot kinematics

### 3.1.1 Denavit-Hartenberg algorithm for the obtention of the direct kinematic model

The step-by-step implementation of the Denavit-Hartenberg algorithm for the ABB IRB 140 is presented here.

To contribute to a better understanding of the process, a drawing of the robot has been generated and each step appropriately depicted.

**D-H 1.** Label the robot links starting from 1 (first free moving robot link) and ending at $n$ (last free moving robot link). The fixed base of the robot will be labeled as 0.

The robot has been dismantled link by link to make visualization easier. Starting from the base, labelled as link 0, there are other 6 robot links.

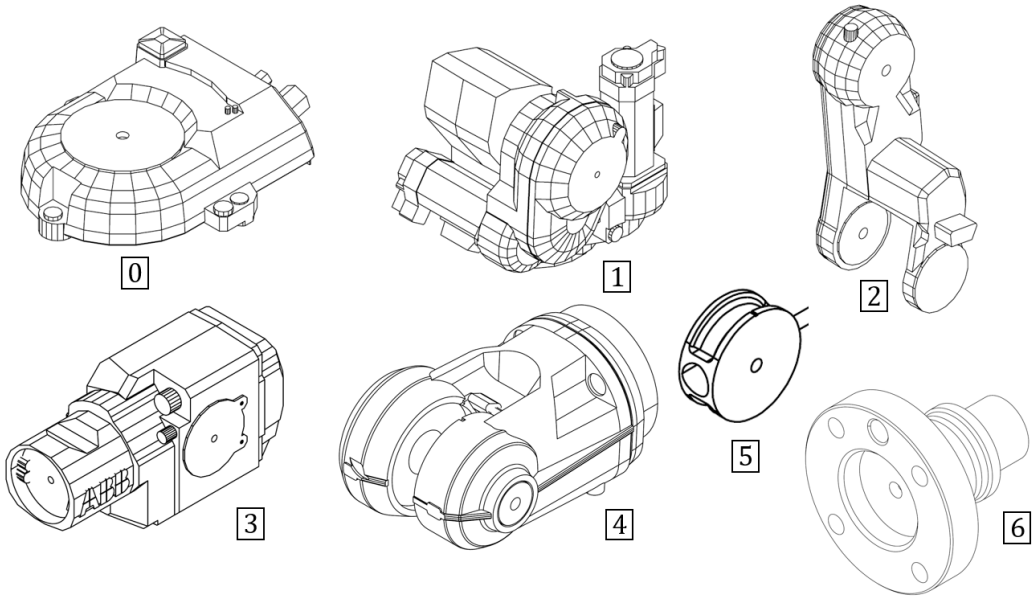



*Figure 3-1. Links of the ABB IRB 140 robot with their individual numeration.*

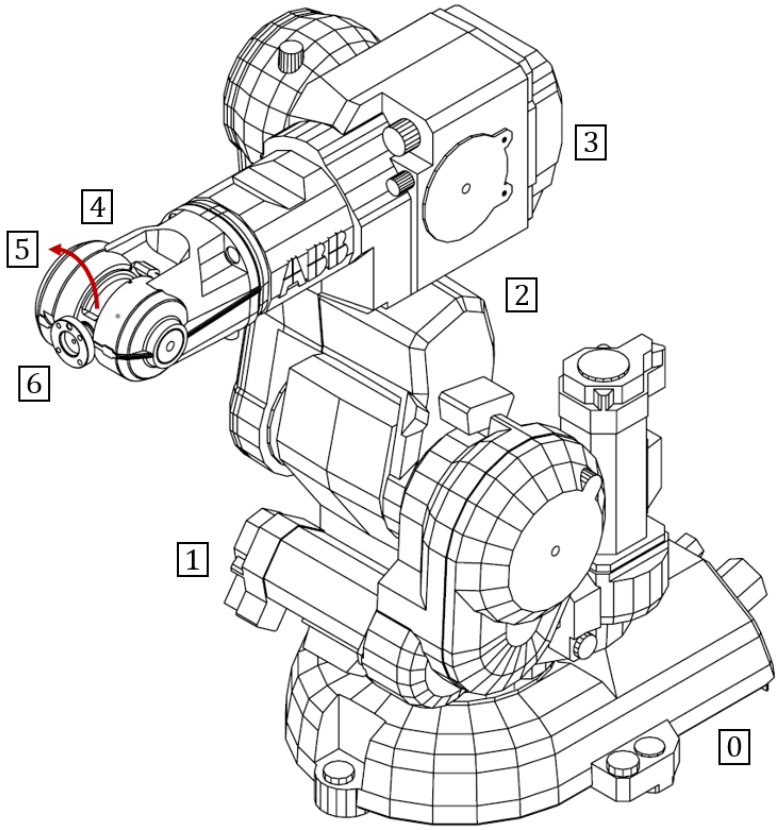



*Figure 3-2. Armed robot ABB IRB 140 with its links numbered.*

**D-H 2.** Label each joint starting from 1 (first degree of freedom) and ending at $n$.

For each joint, there is an associated degree of freedom that is denoted by $\theta_i$, generating the set $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$. They are represented in orange.

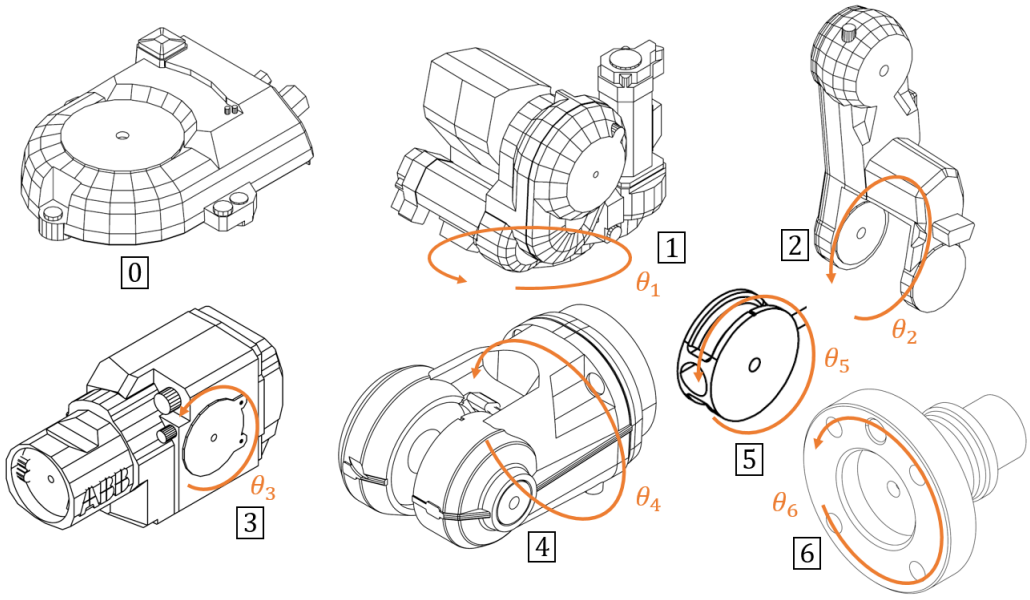



*Figure 3-3. Links of the robot ABB IRB 140 with kinematic pairs represented.*

**D-H 3.** Pinpoint the axis of every joint. If it is rotational, the axis will be its own rotating axis. If it is prismatic it will be the axis along its displacement occurs.

All the joints contain rotating kinematic pairs, whose axis of rotation are drawn.

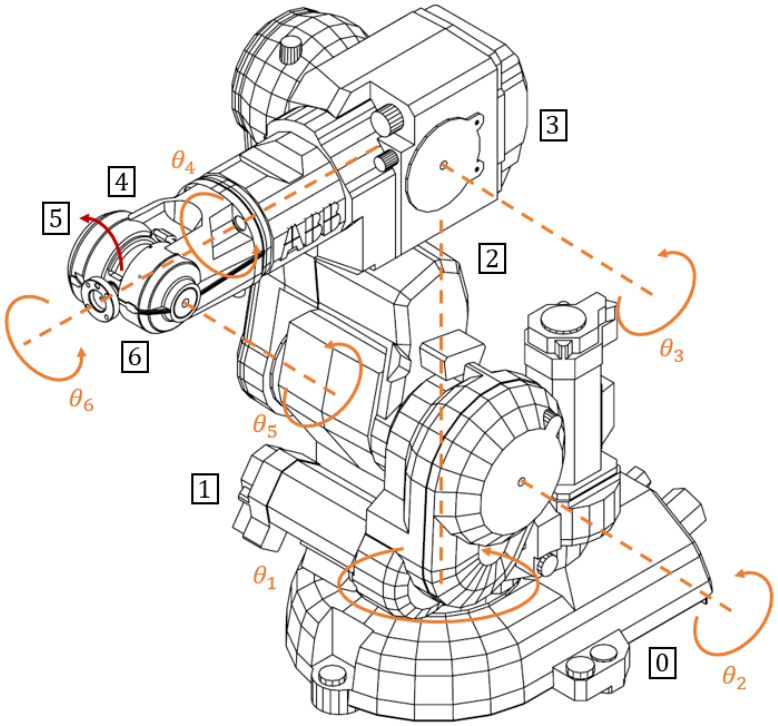



*Figure 3-4. ABB IRB 140 robot with represented rotation axes.*

**D-H 4.** For $i$, from 0 to $n-1$, place the axis $z_i$ where the joint axis $i+1$ stands.

The z axes, represented in indigo, are positioned.

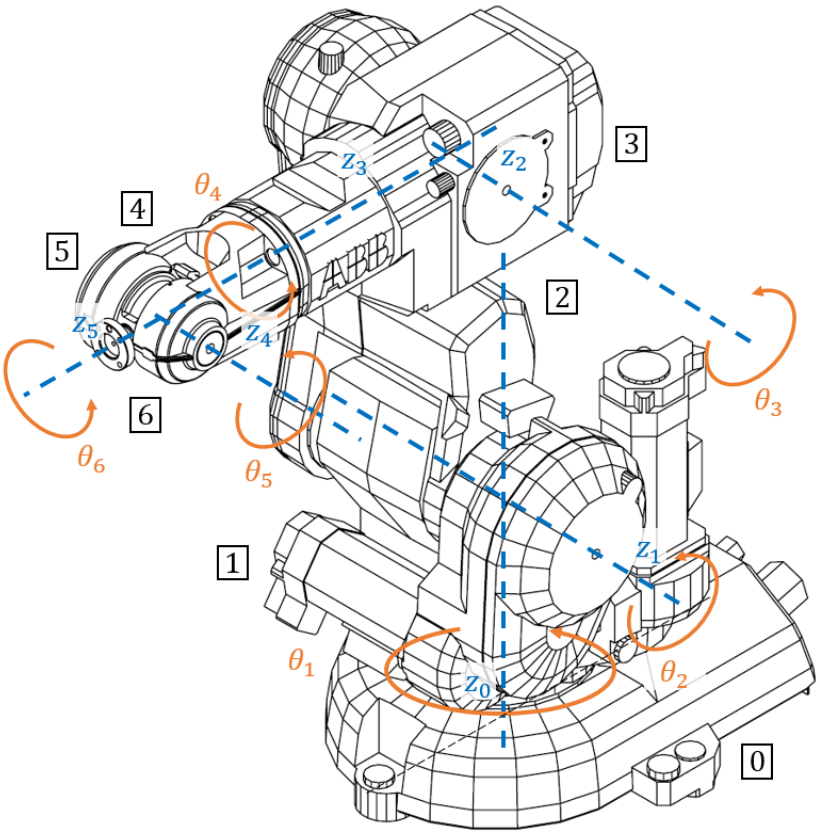



*Figure 3-5. ABB IRB 140 with z axes represented.*

**D-H 5.** Place the origin of the coordinate system $\{S_0\}$ at any point of the axis $z_0$. The axis $x_0$ and $y_0$ will be positioned so that they form a dextrorotatory system with $z_0$.

The $x$ and $y$ axes are represented in red and green respectively. The denominations and origin of each reference system appear in purple.

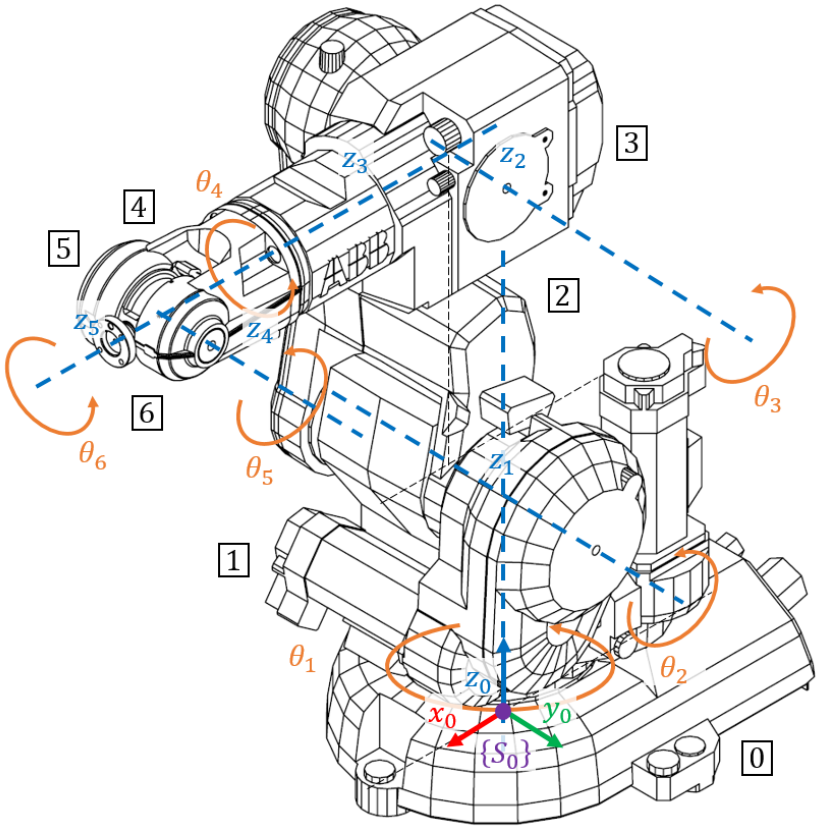



*Figure 3-6. ABB IRB 140 with the coordinate system $\{S_0\}$ represented.*

**D-H 6.** For $i$, from 1 to $n-1$, place the origin of the coordinate system $\{S_i\}$ (attached to link $i$) at the intersection of the axis $z_i$ with the shared perpendicular line of $z_{i-1}$ and $z_i$. If both axes intersected, $\{S_i\}$ would be placed at that intersecting point. If they were parallel $\{S_i\}$ would be place at the joint $i+1$.

In purple points the origins of each system $\{S_i\}$ are represented.

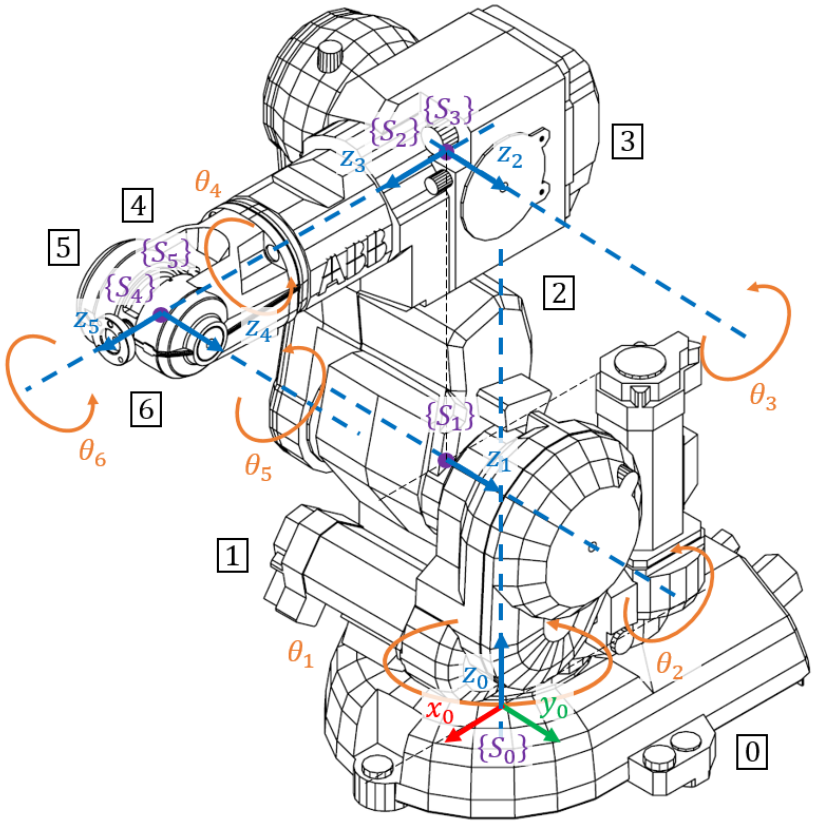



*Figure 3-7. ABB IRB 140 with coordinate systems $\{S_i\}$ represented.*

**D-H 7.** Place $x_i$ at the shared perpendicular line of $z_{i-1}$ and $z_i$.

In red, the axes $x_i$.

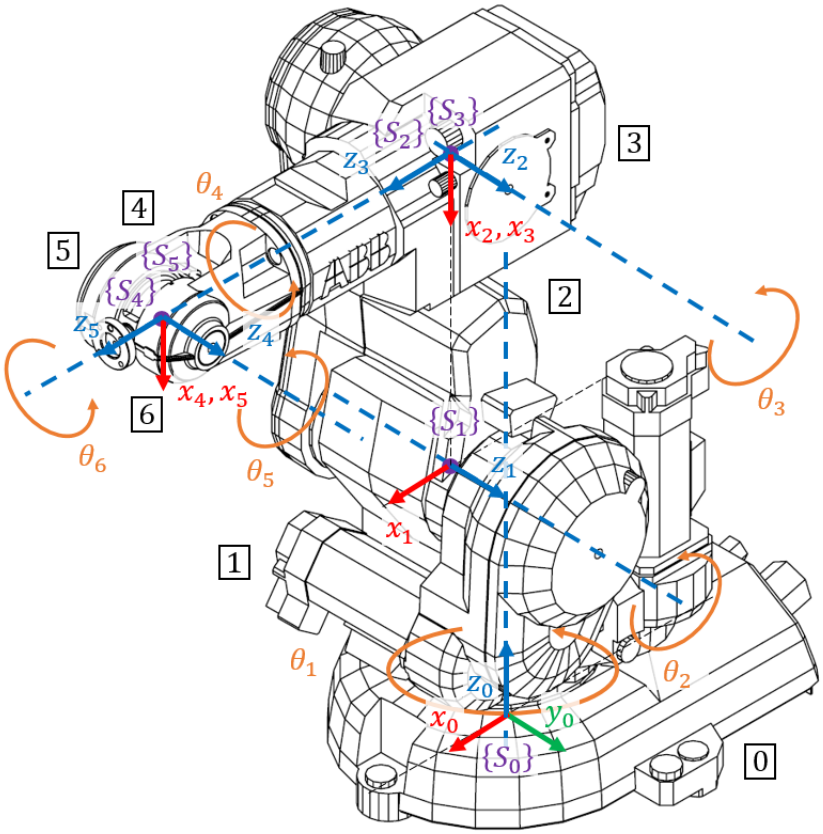



*Figure 3-8. ABB IRB 140 with $x_i$ axes represented.*

**D-H 8.** Place $y_i$ so that it draws a dextrorotatory system with $x_i$ and $z_i$.

After placing all the $y_i$ axes, represented in green, one gets:

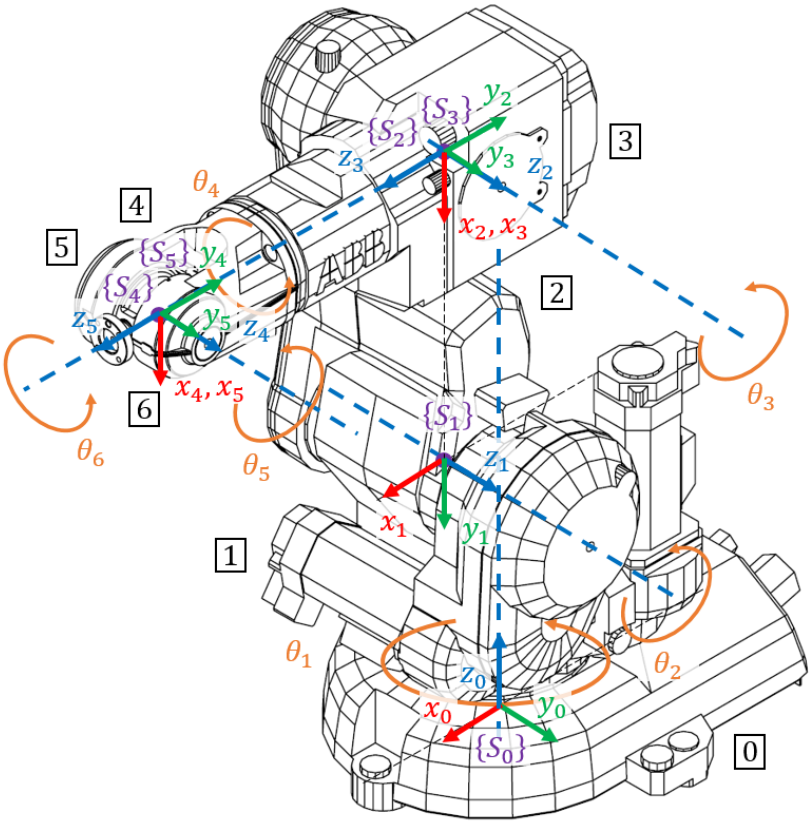



*Figure 3-9. ABB IRB 140 with the $y_i$ axes represented.*

**D-H 9.** Place the coordinate system $\{S_i\}$ at the end effector of the robot, so that $z_n$ matches with the direction of $z_{n-1}$ and $x_n$ is perpendicular to $z_{n-1}$ and $z_n$.

The coordinate system $\{S_6\}$ is established, located at the end of the robot arm, considering a possible tool that would extend longitudinally along the axis $z_5$.

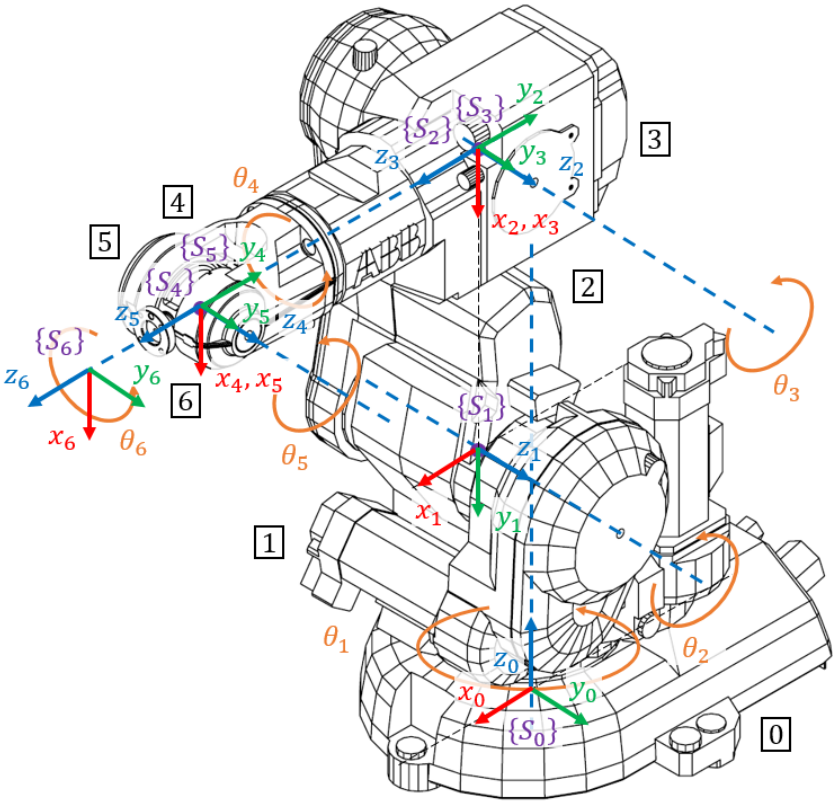



*Figure 3-10. ABB IRB 140 with the coordinate system $\{S_6\}$ represented.*

**D-H 10.** Obtain $\theta_i$ as the angle to be rotated around $z_{i-1}$, so that $x_{i-1}$ and $x_i$ are parallel.

**D-H 11.** Obtain $d_i$ as the distance measured along $z_{i-1}$, that $\{S_{i-1}\}$ would have to be shifted, so that $x_i$ and $x_{i-1}$ are aligned.

**D-H 12.** Obtain $a_i$ as the distance measured along $x_i$ (which would now coincide with $x_{i-1}$) that $\{S_{i-1}\}$ would have to be shifted so that its origin coincides with $\{S_i\}$.

**D-H 13.** Obtain $\alpha_i$ as the angle to be rotated around $x_i$, so that the new $\{S_{i-1}\}$ would coincides with $\{S_i\}$

These four steps (D-H 10 to D-H 13) are performed for $i$, from 1 to 6, and the Denavit-Hartenberg parameters are included in a table. The spatial dimensions have been obtained from the product specifications document of the ABB IRB 140 robot [1].

| Link | $\theta$ [º] | $d$ [$mm$] | $a$ [$mm$] | $\alpha$ [º] |
|---|---|---|---|---|
| **1** | $\theta_1$ | 352 | 70 | $-90$ |
| **2** | $90 + \theta_2$ | 0 | $-360$ | 0 |
| **3** | $\theta_3$ | 0 | 0 | 90 |
| **4** | $\theta_4$ | 380 | 0 | $-90$ |
| **5** | $\theta_5$ | 0 | 0 | 90 |
| **6** | $\theta_6$ | $65 + L_h$ | 0 | 0 |

*Table 3-1. ABB IRB 140 Denavit-Hartenberg parameters. Equivalent to those of [3] [4] [5] [6].*

**D-H 14.** Obtain all transformation matrices ${}^{i-1}A_i$.

These transformation matrices include each of the particular transformations to which the reference systems are subjected, these are: a rotation around the axis $z_{i-1}$ of an angle $\theta_i$, a translation along $z_{i-1}$ of a distance $d_i$, a translation along $x_i$ of a distance $a_i$ and a rotation about the axis $x_i$ of an angle $\alpha_i$.

$$ {}^{i-1}A_i = \begin{bmatrix} c(\theta_i) & -c(\alpha_i)s(\theta_i) & s(\alpha_i)s(\theta_i) & a_i c(\theta_i) \\ s(\theta_i) & c(\alpha_i)c(\theta_i) & -s(\alpha_i)c(\theta_i) & a_i s(\theta_i) \\ 0 & s(\alpha_i) & c(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.1} $$

**D-H 15.** Obtain the transformation matrix that relates the base coordinate system $\{S_0\}$ with the end effector coordinate system $\{S_6\}$: $T = {}^0A_1 \cdot {}^1A_2 \cdot \ldots \cdot {}^{n-1}A_n$.

With a total of 6 joints, the homogeneous transformation matrix from the origin of the reference system $\{S_0\}$ to the origin of the reference system of the end effector $\{S_6\}$ is:

$$ T = {}^0A_1 \cdot {}^1A_2 \cdot {}^2A_3 \cdot {}^3A_4 \cdot {}^4A_5 \cdot {}^5A_6 \tag{3.2} $$

**D-H 16.** This transformation matrix $T$ generates a rotation matrix and a translation vector as function of every joint coordinate.

This relation between the orientation and position of the end effector and joint coordinates, is what forms the foundations of kinematic modeling.

Implementation of the Denavit-Hartenberg was programed in Code 3.

### Jacobian matrix

With a $7 \times 6$ dimension, it is declared and calculated in Code 5. It has the form:

$$ J = \underbrace{\begin{bmatrix} \frac{\partial f_x}{\partial q_1} & \cdots & \frac{\partial f_x}{\partial q_6} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_d}{\partial q_1} & \cdots & \frac{\partial f_d}{\partial q_6} \end{bmatrix}}_{7\times 6} \tag{3.3} $$

#### *3.1.1.1 Simulink direct kinematic model subsystem*

The inputs of this subsystem are the current joint angles, velocities and accelerations. These are processed separately in three different function blocks, one to obtain the equivalent current pose, another to obtain the current pose velocity and finally a block to obtain the current pose acceleration. These three are the outputs of the subsystem.

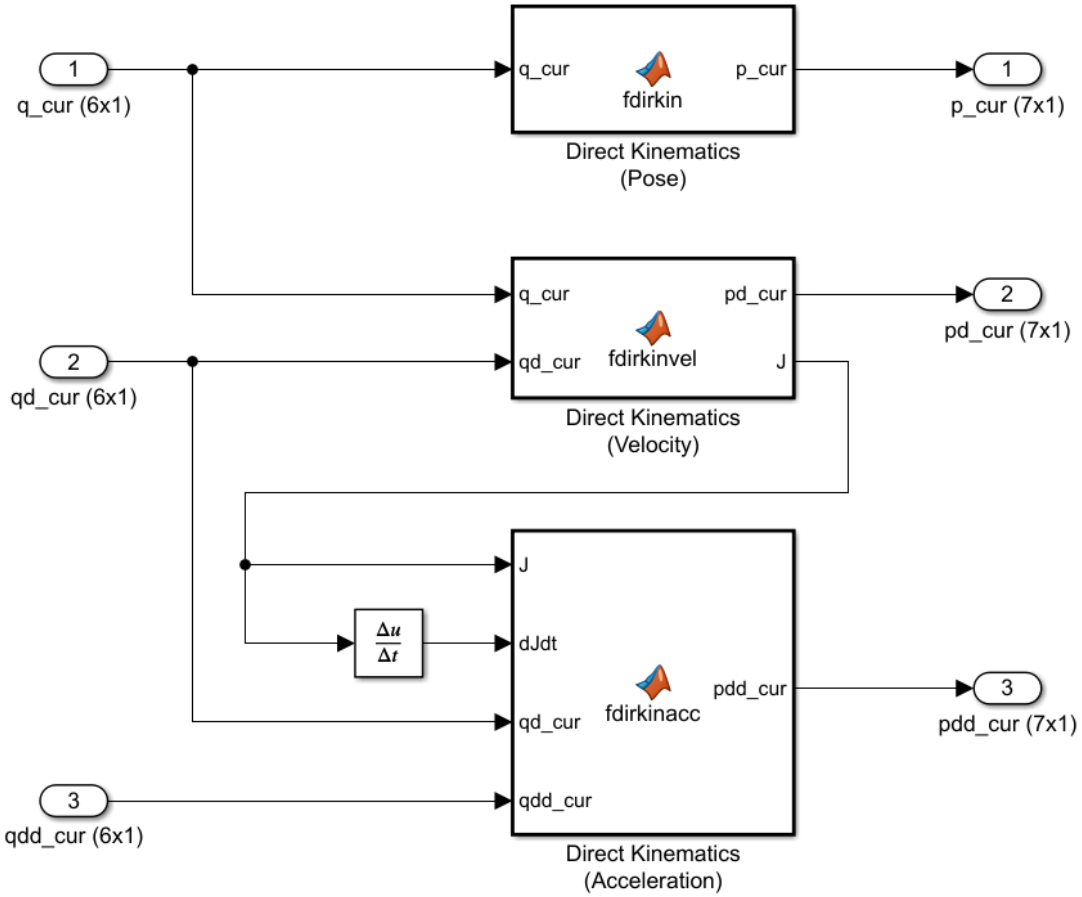



*Figure 3-11. Direct kinematic model subsystem.*

To observe the internal code of each function block, they can be found at the code appendix: Code 4, Code 6 and Code 7.

### 3.1.2 Validation of the direct kinematic model

The validation method proposed for this direct kinematic model code is the comparison with the results of RobotStudio by inserting arbitrary joint angles. By means of the FlexPendant tool, it is possible to simulate a direct control of the robot with the help of a virtual control panel, emulating the panel of the real robot.

A new solution is created with a station and a controller, and the ABB IRB 140 is selected in the *Robot model* tab.

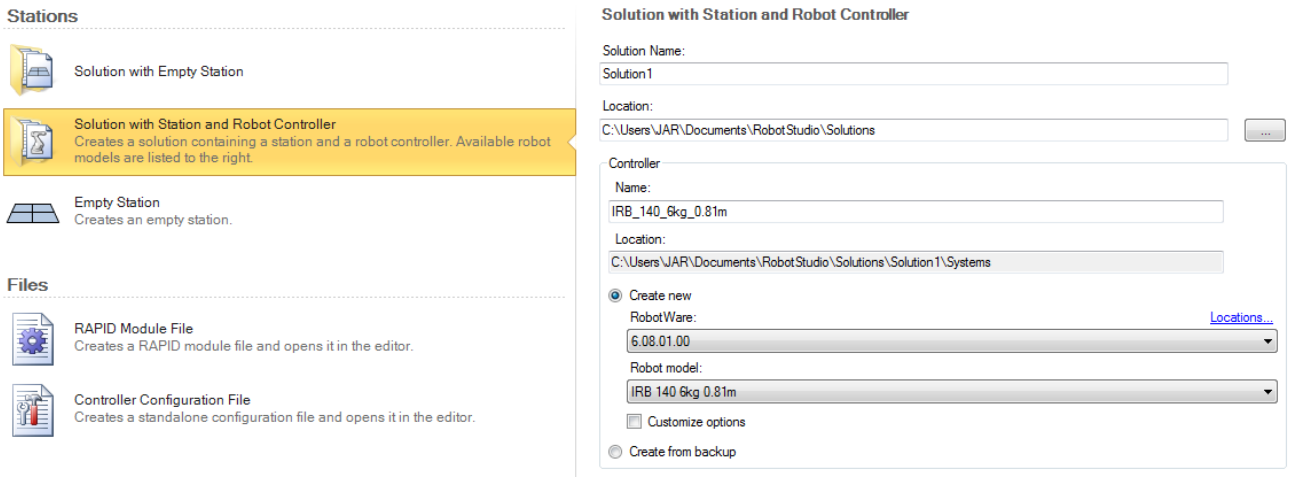



*Figure 3-12. New solution menu on RobotStudio.*

At the main window, clicking on *Controller*, the button that starts the manual controller in FlexPendant can be found.

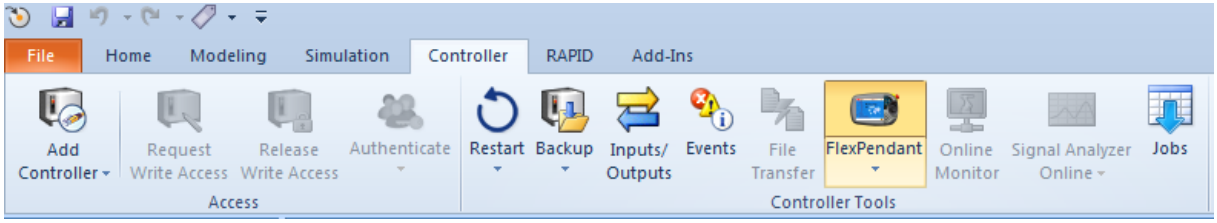



*Figure 3-13. RobotStudio menu with FlexPendant highlighted.*

Three independent tests have been generated with different random joint angle values and introduced in both RobotStudio and the MATLAB function (Code 4).

#### *3.1.2.1 Test no. 1*

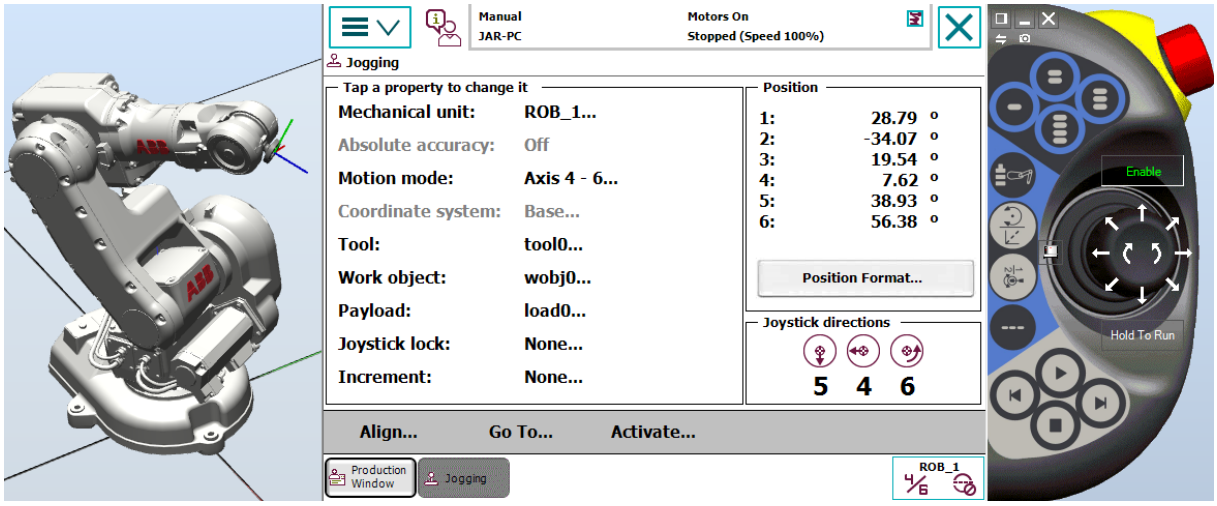



*Figure 3-14. Test no. 1. Joint angles.*

```
q0 = deg2rad([28.79 -34.07 19.54 7.62 38.93 56.38]');
```

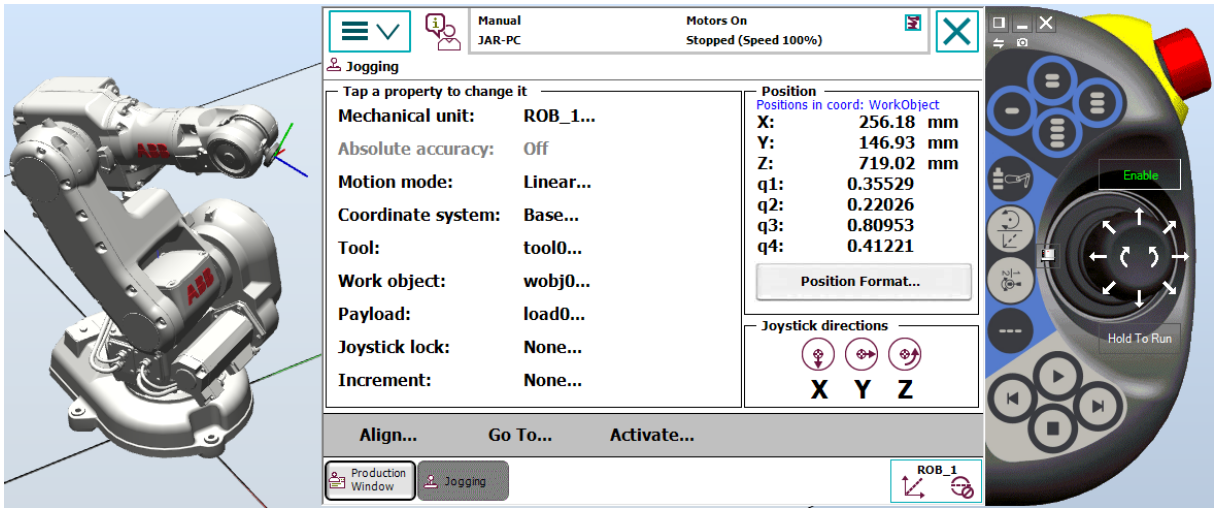



*Figure 3-15. Test no. 1. Pose.*

The results of the function are:

```
p_cur =

   0.256169020979862
   0.146951811198975
   0.719041733432580
   0.355266700974361
   0.220219321995462
   0.809549214980808
   0.412212433003434
```

### *3.1.2.2 Test no. 2*

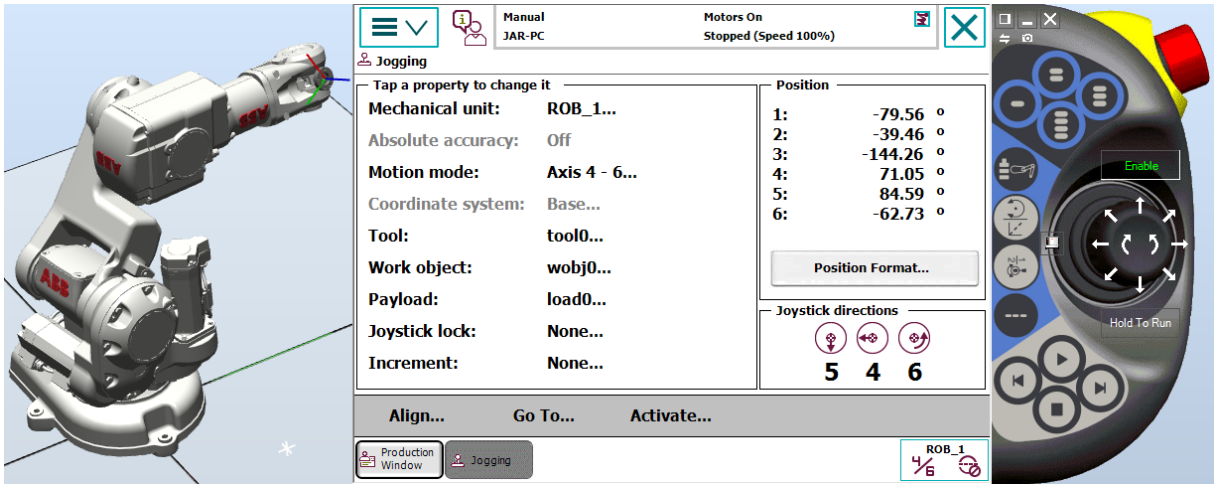



*Figure 3-16. Test no. 2. Joint angles.*

```
q0 = deg2rad([-79.56 -39.46 -144.26 71.05 84.59 -62.73]');
```

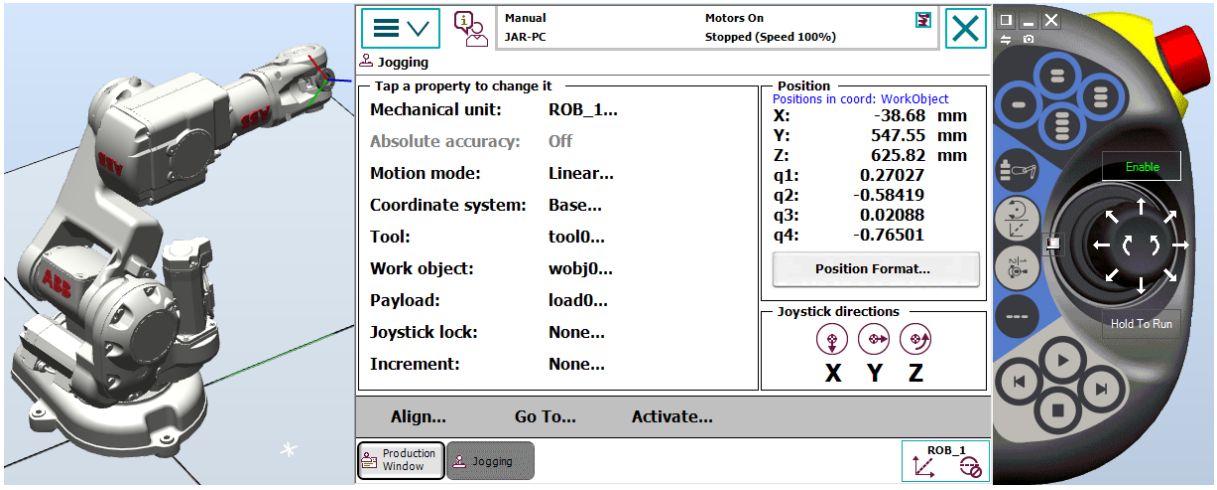



*Figure 3-17. Test no. 2. Pose.*

The results of the function are:

```
p_cur =

  -0.038652641991809
   0.547532574301628
   0.625862378140839
   0.270294220765169
  -0.584219321953463
   0.020946045331934
  -0.764977176955064
```

### *3.1.2.3 Test no. 3*

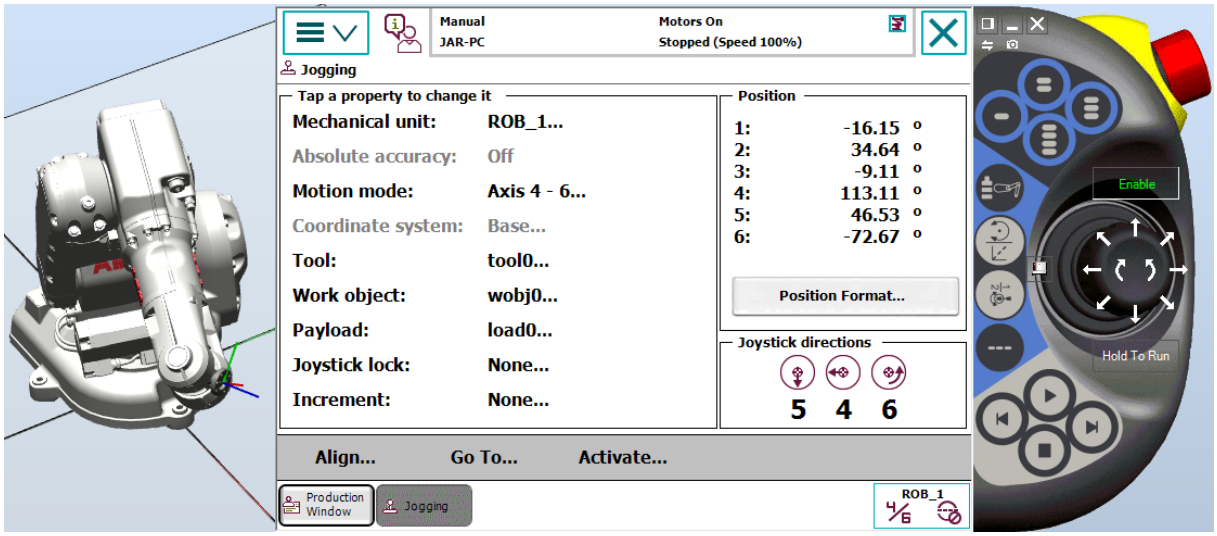



*Figure 3-18. Test no. 3. Joint angles.*

```
q0 = deg2rad([-16.15 34.64 -9.11 113.11 46.53 -72.67]');
```

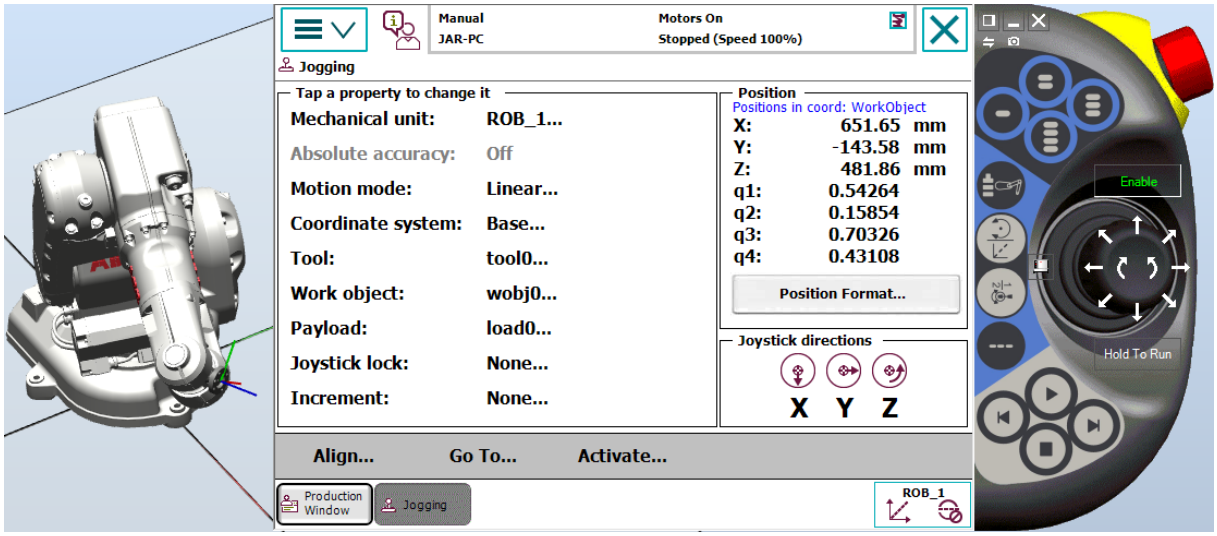



*Figure 3-19. Test no. 3. Pose.*

And finally, the results of the function in this test are:

```
p_cur =

   0.651650430407853
  -0.143535605813641
   0.481847014812847
   0.542604000206256
   0.158528784053503
   0.703280404944469
   0.431098823482308
```

Hence, the function is validated and prepared for later use.

### 3.1.3 Obtention of the inverse kinematic model

The homogeneous transformation matrix $T$ serves for basis as well for the inverse kinematic model. This time, joint coordinates are the unknown variables and the goal is to reach a certain pose state. A system of 12 non-linear equation is generated (9 from the rotation matrix and 3 from the translation vector) with 6 variables $(q_1, q_2, q_3, q_4, q_5, q_6)$, so that 6 of those 12 equations are linearly dependent.

$$\begin{matrix} x = f_x(q_1, \dots, q_6) & y = f_y(q_1, \dots, q_6) & z = f_z(q_1, \dots, q_6) \\ R_{11} = f_{R_{11}}(q_1, \dots, q_6) & R_{12} = f_{R_{12}}(q_1, \dots, q_6) & \dots \quad R_{33} = f_{R_{33}}(q_1, \dots, q_6) \end{matrix} \tag{3.4}$$

This situation implies that the non-linear equation system must be solved through numerical methods, instead of preconstructed algorithms. The selected method is the MATLAB function `fsolve`. It reaches one of the multiple solutions this problem has.

Another issue the robot would face is the limited range of movements all joints suffer. (Table 3-2) This limitation ought to be included in the model, but the function `fsolve` does not admit solution boundaries. However, there will not be any problem derived from this, since the final movement tests at the end of this master's thesis will not reach these limit angles, although any hypothetical work extension should take these into account.

| Type of motion | Range of movement |
|---|---|
| Axis 1: Rotation motion | + 180° to - 180° |
| Axis 2: Arm motion | + 110° to - 90° |
| Axis 3: Arm motion | + 50° to - 230° |
| Axis 4: Wrist motion | + 200° to - 200° Default<br>+ 165 revolutions to - 165 revolutions Max. [i] |
| Axis 5: Bend motion | + 115° to - 115° |
| Axis 6: Turn motion | + 400° to - 400° Default<br>+ 163 revolutions to -163 revolutions Max. *i* |

*Table 3-2. Joint angle limits for the ABB IRB 140. [1]*

A similar scenario happens with the joint velocity limits, although the kinematics of the robot make it simpler: It exceeds joint velocity limits or not. These neither have been implemented for the sake of simplicity (Table 3-3).

| Axis No. | IRB 140-6/0.8 | IRB 140T-6/0.8 |
|---|---|---|
| 1 | 200°/s | 250°/s |
| 2 | 200°/s | 250°/s |
| 3 | 260°/s | 260°/s |
| 4 | 360°/s | 360°/s |
| 5 | 360°/s | 360°/s |
| 6 | 450°/s | 450°/s |

*Table 3-3. Joint velocity limits for a 3-phase power supplied ABB IRB 140. [1]*

The Simulink schematic of the inverse kinematics model subsystem is:

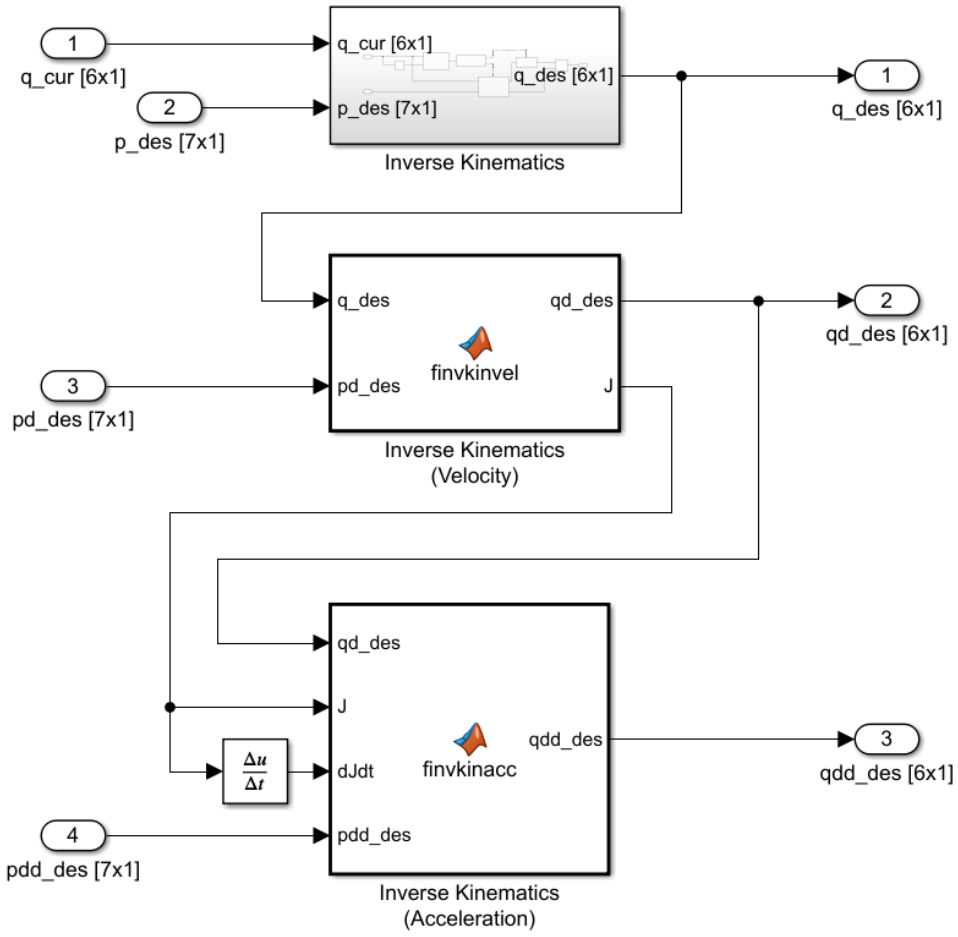



*Figure 3-20. Inverse kinematics model subsystem.*

A mechanism is designed for the inverse kinematics subsystem for pose calculation, which allows not to recalculate new joint coordinates if the pose remains unchanged, saving this way computation time. The mechanism is as follows:

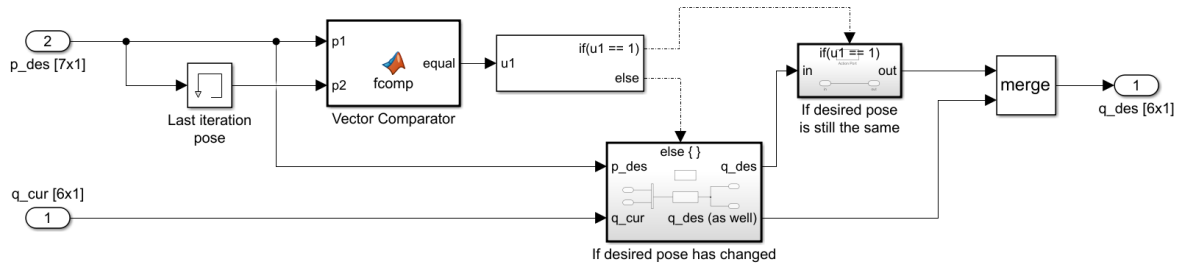



*Figure 3-21. Anti-recalculation mechanism.*

The current pose and that of the previous time step are compared (Code 8) if they are not the same a new joint coordinates vector $q$ is calculated. On the other hand, if they are equal, it carries the last calculated $q$ to the output without processing it again.

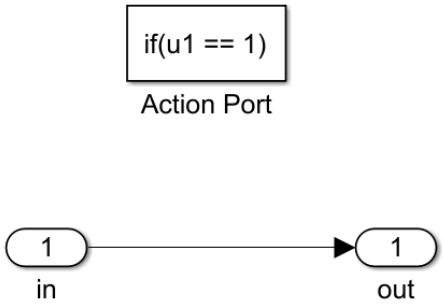



*Figure 3-22. If both pose vectors are equal subsystem.*

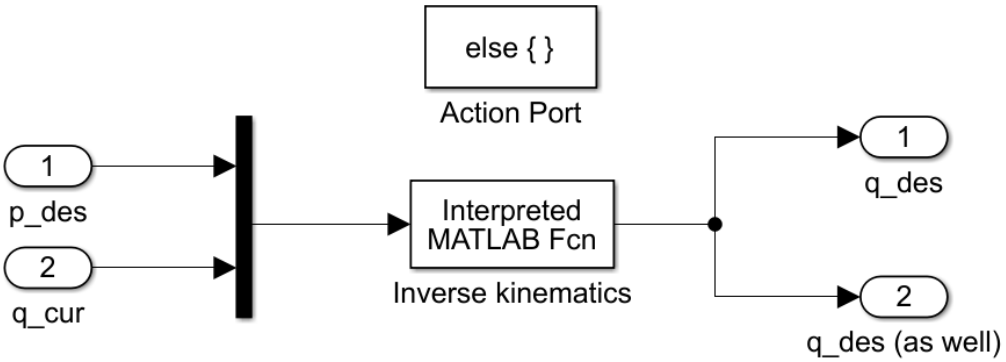



*Figure 3-23. Else (both pose vectors are not equal) subsystem.*

The scripts for the inverse kinematic model are Code 9, Code 10 and Code 11.

### 3.1.4 Validation of the inverse kinematic model

The same 3 tests used at section 3.1.2 have been performed again on this task and its results are shown below.

#### *3.1.4.1 Test no. 1*

Entering the input vector, featuring pose coordinates and starting joint angles:

```
input = [0.256169020979862;
         0.146951811198975;
         0.719041733432580;
         0.355266700974361;
         0.220219321995462;
         0.809549214980808;
         0.412212433003434;
         0;
         0;
         0;
         0;
         0;
         0];
```

The resulting joint angles (converted to degrees) necessary to reach them are the following:

```
q_des =

  28.789999998961303
 -34.070000000785029
  19.540000000894274
   7.620000001600972
  38.930000000019035
  56.379999998312144
```

#### *3.1.4.2 Test no. 2*

Pose coordinates:

```
input = [-0.038652641991809
         0.547532574301628
         0.625862378140839
         0.270294220765169
         -0.584219321953463
         0.020946045331934
         -0.764977176955064
         0;
         0;
         0;
         0;
         0;
         0];
```

Joint angles (converted to degrees):

```
q_des =

  -79.5600000000176
  -95.3067957443123
  -35.7399999982214
   82.6735661715858
   71.6844008779341
  -115.1240977575642
```

This time the solver reaches a different solution although perfectly valid, since after introducing these values back into the direct kinematics function, it shows the initial pose coordinates again.

#### *3.1.4.3 Test no. 3*

Pose coordinates:

```
input = [0.651650430407853
         -0.143535605813641
         0.481847014812847
         0.542604000206256
         0.158528784053503
         0.703280404944469
         0.431098823482308
         0;
         0;
         0;
         0;
         0;
         0];
```

Joint coordinates.

```
q_des =

  -16.1499999999874
   34.6399999999286
  -9.1099999998241
  -66.8899999998916
  -46.5300000000301
   107.3299999998540
```

Once again, the joint coordinates do not match with their original values, showing another geometrical possibility to reach the pose.

All tests pass, validating the inverse kinematics function.

## 3.2 Robot dynamics

### 3.2.1 Calculation of masses and inertia of the links

Each of the links of the robot has a particular geometry, which with the combination of internal elements (actuators, brakes, wiring) generates a highly complex three-dimensional model.

CAD models of this complex depiction are not published by ABB due to reasons of confidentiality, nor are the masses of each link. That is why the decision to obtain the masses and inertia of the links from the simplified solid CAD model was made.

Starting from a known fact, which the total mass of the robot, 98 kg, and assuming a constant and homogeneous density, it is possible to make a proportional estimate of the mass of each of the links. SolidWorks allows the user to know the volume of the links. The proportion of an element's volume with respect to the robot's total volume is multiplied by the total mass to offer an estimated link mass value.

$$m_{link} = m_{total} \cdot \frac{v_{link}}{v_{total}} \tag{3.5}$$

| Link | Volume $[mm^3]$ | Proportional mass $[kg]$ |
|---|---|---|
| Base | 12405874,57 | 26,3635884 |
| 1 | 16262913,55 | 34,56013977 |
| 2 | 7528223,81 | 15,99814611 |
| 3 | 7989940,24 | 16,97933464 |
| 4 | 1759858,35 | 3,739855737 |
| 5 | 143038,24 | 0,303969 |
| 6 | 25865,43 | 0,054966342 |
| | | |
| Sum | 46115714,19 | 98 |

*Table 3-4. Estimated link masses.*

### 3.2.2 Modeling of the ABB IRB 140 robot in SimScape Multibody

#### *3.2.2.1 Preparation of the CAD models of the links prior to insertion*

A physical model of the robot arm is assembled, based on the CAD models of each of the links, which are downloadable on the ABB website.

The origins of the coordinate systems of every CAD model have had to be repositioned, making then coincide with those of the Denavit-Hartenberg algorithm. SolidWorks has been used in this task.

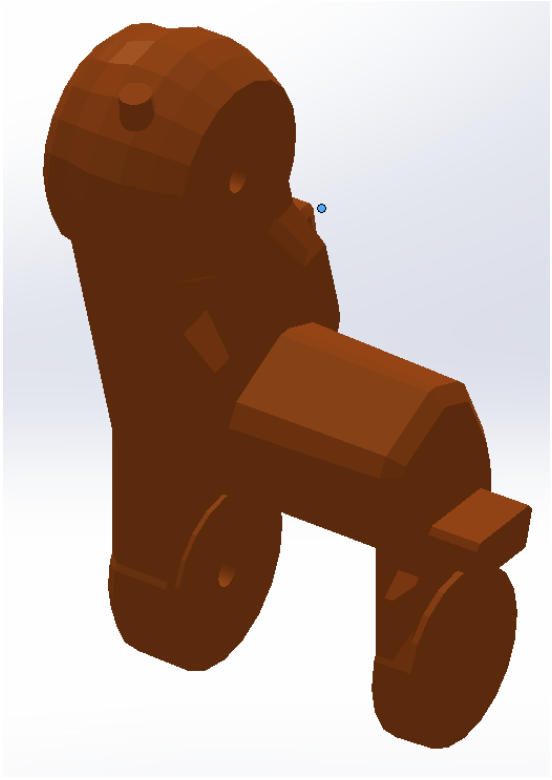


*Figure 3-24. Example of a modified origin of coordinate system. Link 2 with* $\{S_2\}$

#### *3.2.2.2 Inicialization of the SimScape Multibody model*

The model is underlied by three initialization blocks. The first declares the solver to be executed in the numerical resolution of the mechanism and the modifiable parameters associated with it. The second block denotes the origin of global coordinates. Finally, the third defines the configuration of the mechanism: the gravity vector and the linearization delta.


*Figure 3-25. SimScape model initialization blocks: Solver configuration, Origin declaration and Mechanism configuration.*

### *3.2.2.3 Loading and inserting the CAD models of the links*

The block used to load the link CAD models is the File Solid block. It allows to modify its mass, in order to automatically establish its inertia around the local coordinate system that was modified in 3.2.2.1.

The estimated mass is entered numerically on the corresponding tab without calling it from the Workspace, because doing so slows down the execution significantly.

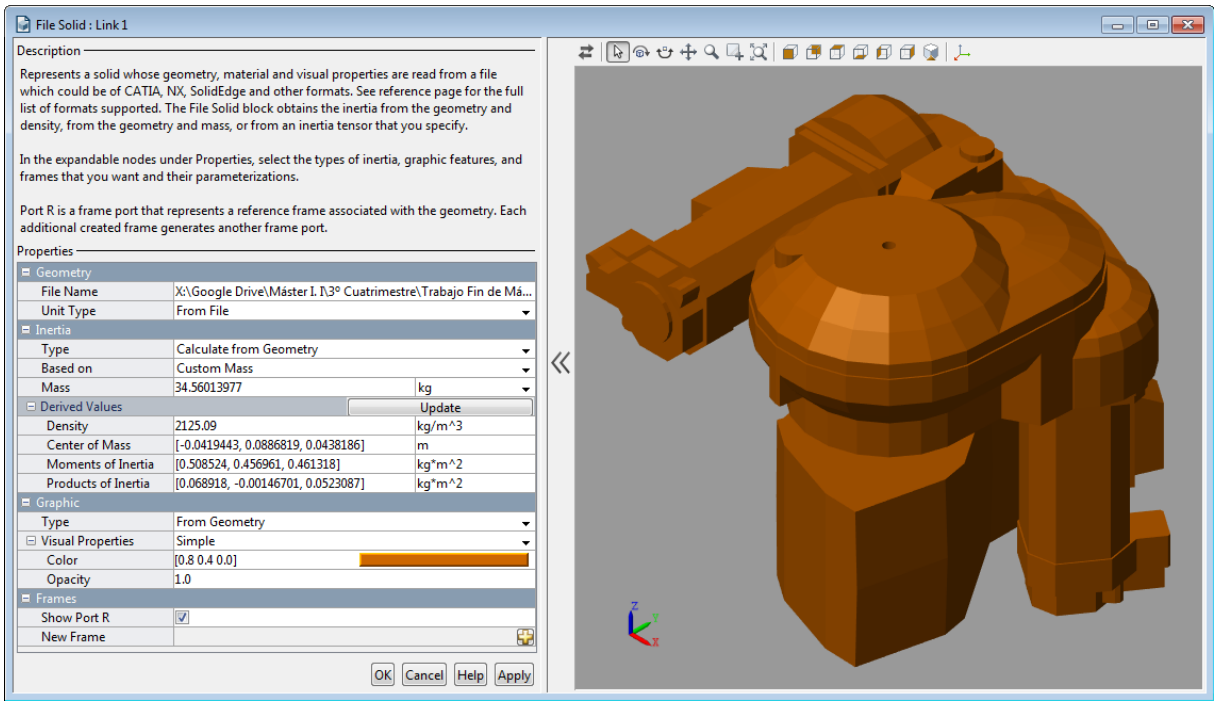



*Figure 3-26. File Solid block for Link 1.*

### *3.2.2.4 Transformation of coordinate systems*

Each set of translation/rotation Rigid Transform blocks transforms the current local coordinate system into the next coordinate system of the Denavit-Hartenberg algorithm.

Each individual block represents a rotation or translation, never both at the same time, in order to avoid executions in an undesired order and facilitate a better understanding of the block algebra.

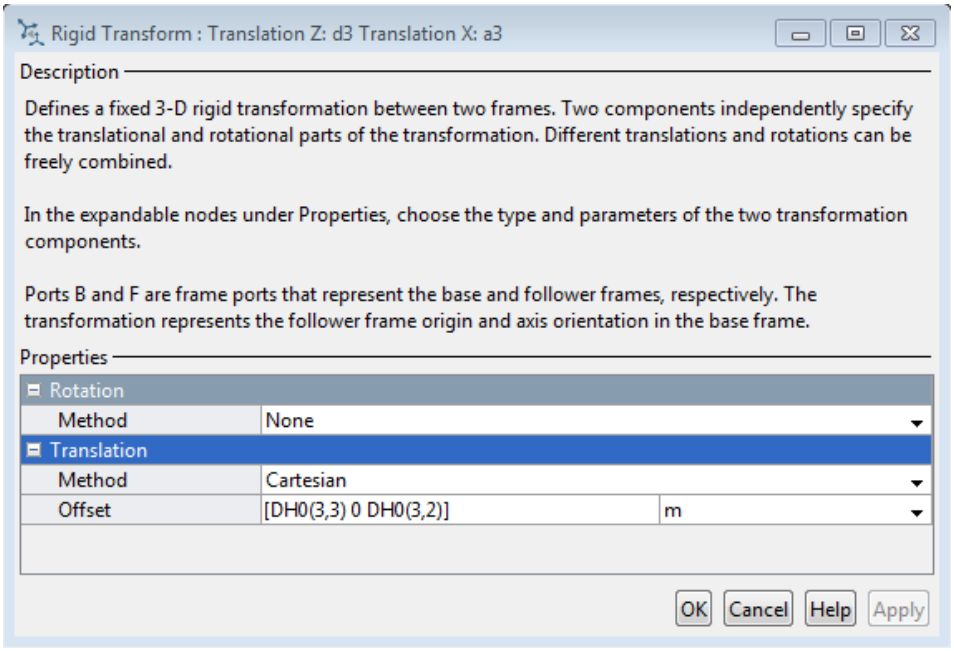



*Figure 3-27. Rigid transform block for translation along* $z_3$.

### *3.2.2.5 Joint declaration*

Ultimately, the Revolute Joint block generates an articulation around the z axis of a local coordinate system. All robot joints can be modelled with this block, for all joints are rotational.

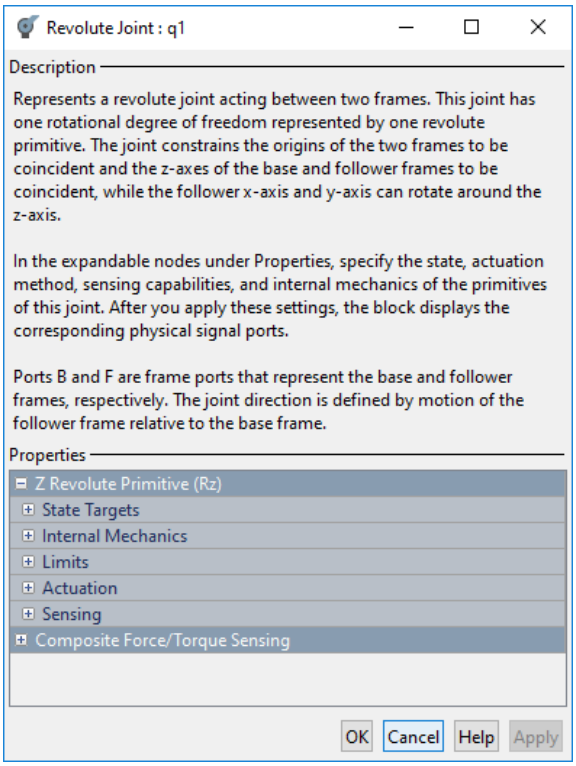



*Figure 3-28. Revolute Joint block.*

Between the many options available in this block, the most important and used in this paper are the physical inputs to be put in and those that should be obtained at the end.

#### *3.2.2.6 Torque limitations*

Like every real system, inputs are limited to a specific range. Joint electric motors can only provide up to a certain amount of torque in one direction or another. The rate at which these torques change is also limited. However, the only information available at the product specification document is the maximum torques at joints 4, 5 and 6.

| Robot type | Max wrist torque axis 4 and 5 | Max wrist torque axis 6 | Max torque valid at load |
|---|---|---|---|
| IRB 140(T)-6/0.8 | 8.58 Nm | 4.91 Nm | 5 kg |

*Table 3-5. Maximum torques at joints 4, 5 and 6. [1]*

These have been included in the model as Saturation and Rate Limiter blocks, whose numerical values are called from Workspace.

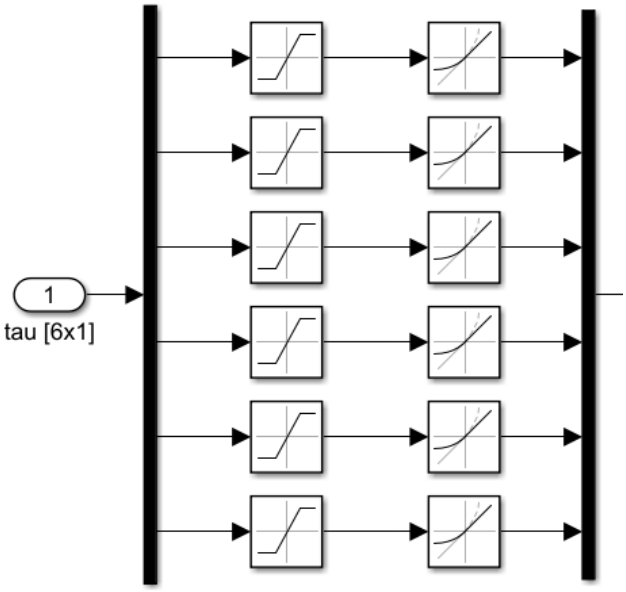



*Figure 3-29. Saturation and Rate Limiter blocks for every joint motor.*

Given that the actuator features are confidential, modelling these from scratch would represent an arduous task. Thus, approximate values for their constraints have been assumed:

| Joint actuator | Saturation limits $[N \cdot m]$ | Slew rate limits $[N \cdot m/s]$ |
|---|---|---|
| 1 | $[-150,150]$ | $[-10^5, 10^5]$ |
| 2 | $[-150,150]$ | $[-10^5, 10^5]$ |
| 3 | $[-150,150]$ | $[-10^5, 10^5]$ |
| 4 | $[-8.58,8.58]$ | $[-10^5, 10^5]$ |
| 5 | $[-8.58,8.58]$ | $[-10^5, 10^5]$ |
| 6 | $[-4.91,4.91]$ | $[-10^5, 10^5]$ |

*Table 3-6. Assumed saturation and slew rate limits for all joint actuators.*

Then these non-linear limited torques are introduced in the revolute joint blocks, which in the proper D-H configuration that defines the robot spatially, throw the current vectors of joint angles ($q$), joint velocities ($\dot{q}$) and joint accelerations ($\ddot{q}$).

### *3.2.2.7 ABB IRB 140 full system diagram*

The whole system is presented in Figure 3-30. Its inputs are torques and its outputs are joint angles, joint velocities and joint accelerations.

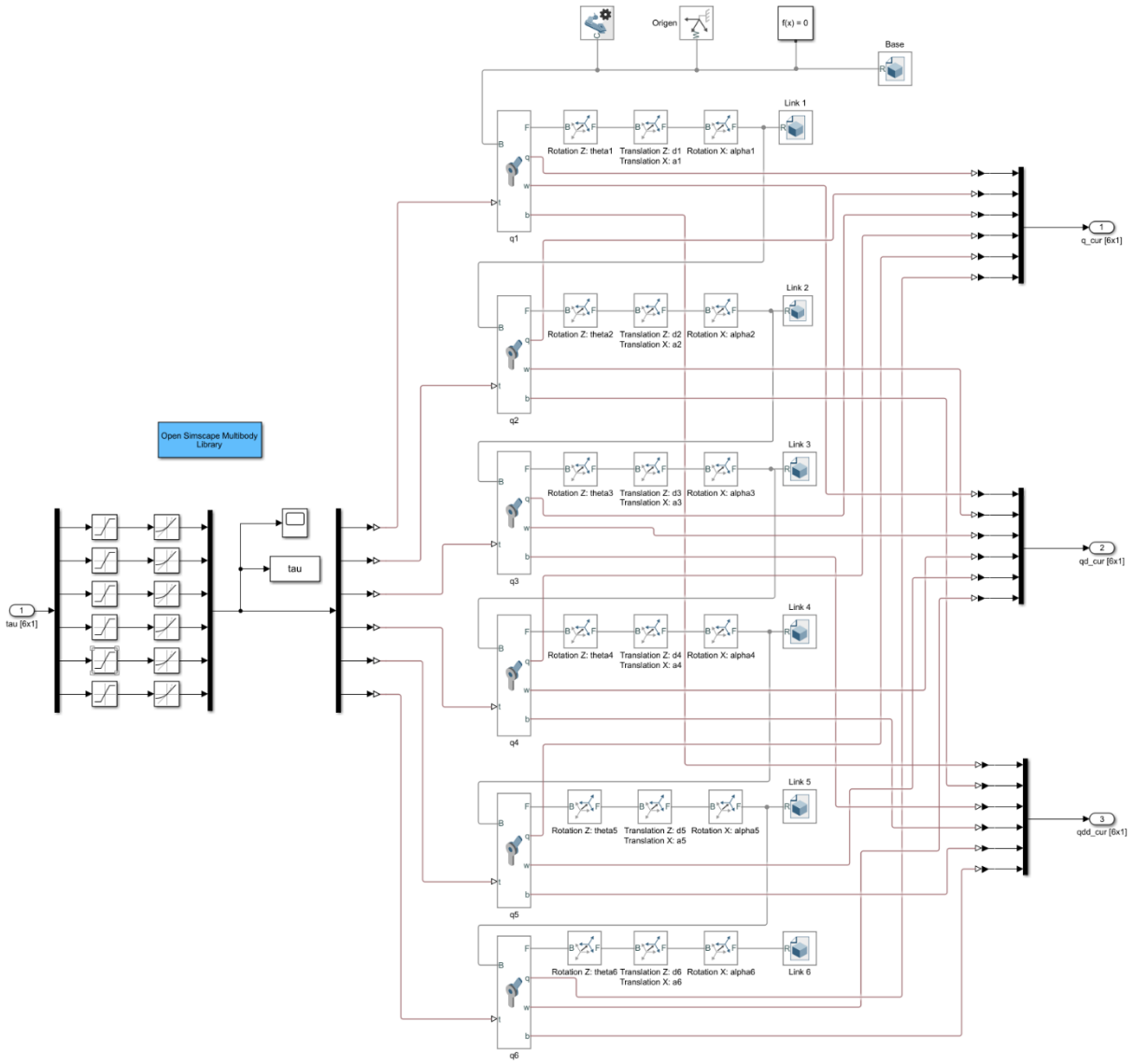



*Figure 3-30. ABB IRB 140 full system diagram.*

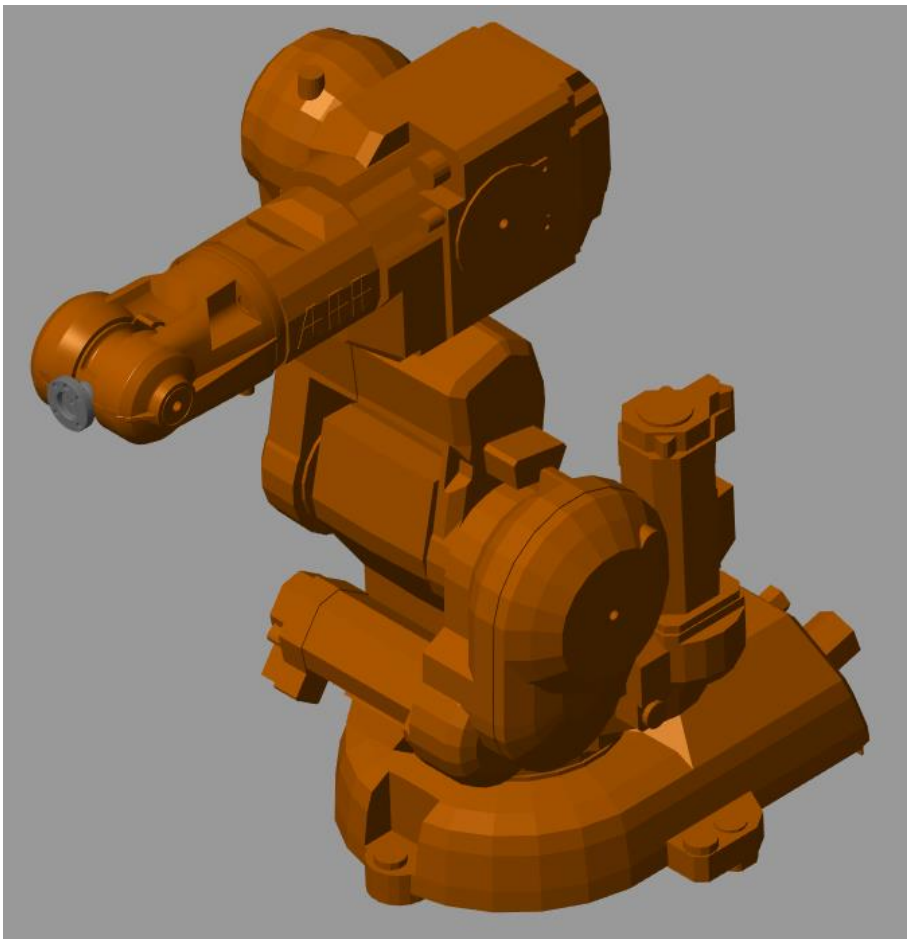


*Figure 3-31. 3-D representation of the ABB IRB 140 on MATLAB.*

### *3.2.2.8 Model validation*

The robot model was tested in order to verify that the end effector positions itself where it must. The same joint angles of the 3 tests of 3.1.2 have been used. After the introduction of a column vector of joint angles in degrees, they are converted into radians by the function `deg2rad` and fed into the ABB IRB 140 SimScape model block.

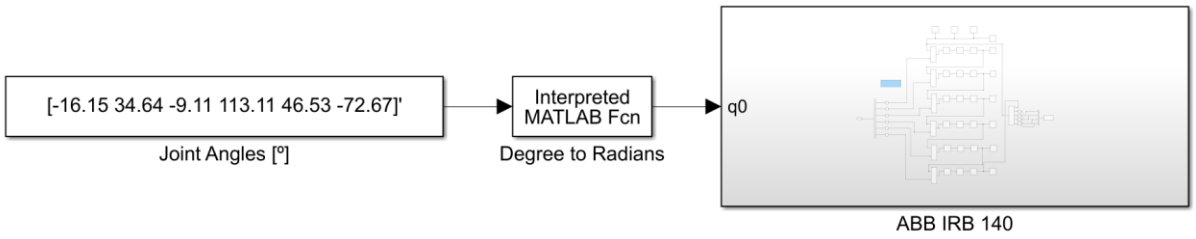



*Figure 3-32. Main schematic for ABB IRB 140 model testing.*

This robot block has been modified to accept joint angles as input. A new SimScape-specific block, the *Transform Sensor* block, was placed setting the base reference as the origin and the frame reference as the end effector. The resulting pose of the end effector is loaded this way onto the Workspace.

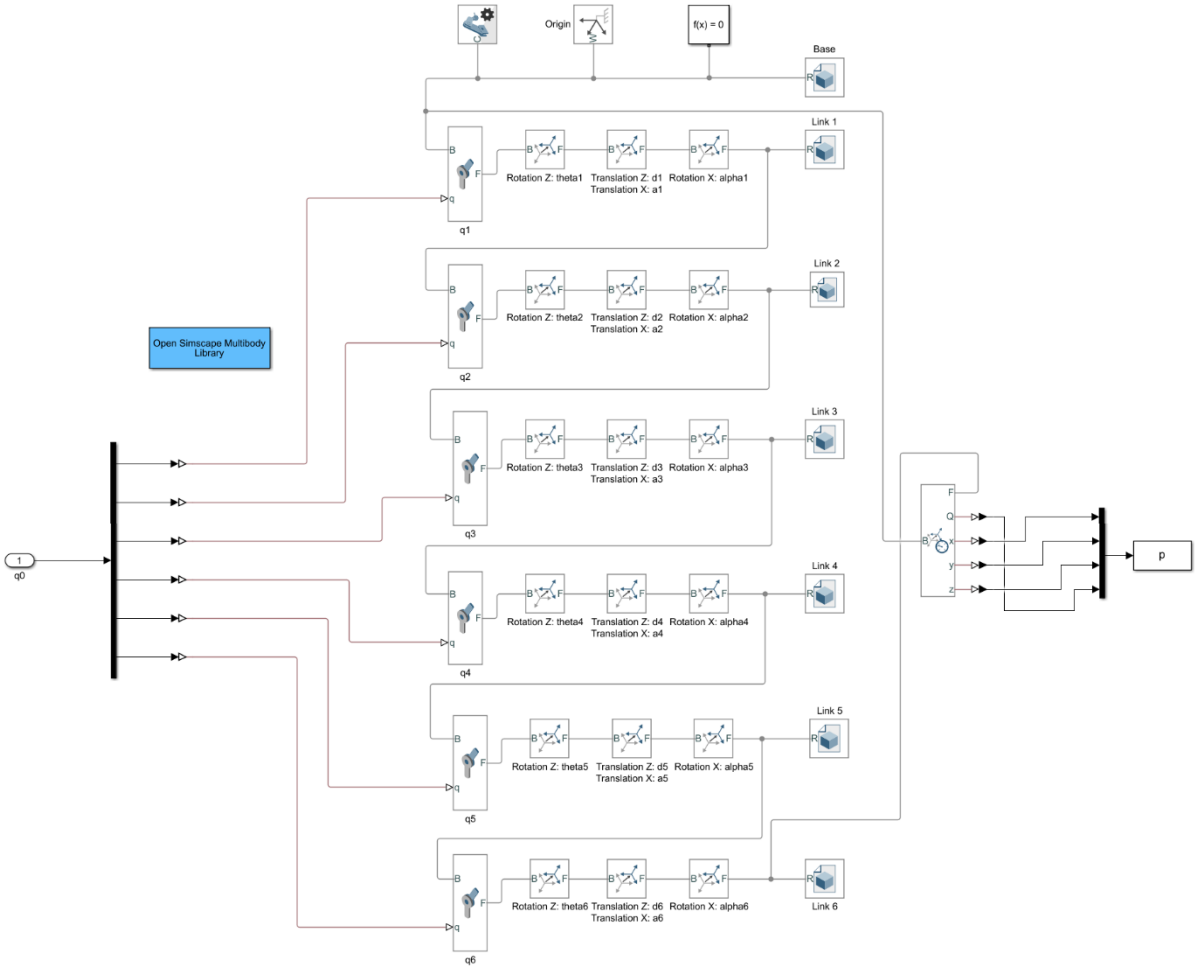



*Figure 3-33. ABB IRB 140 schematic for model testing.*

- **Test no. 1**

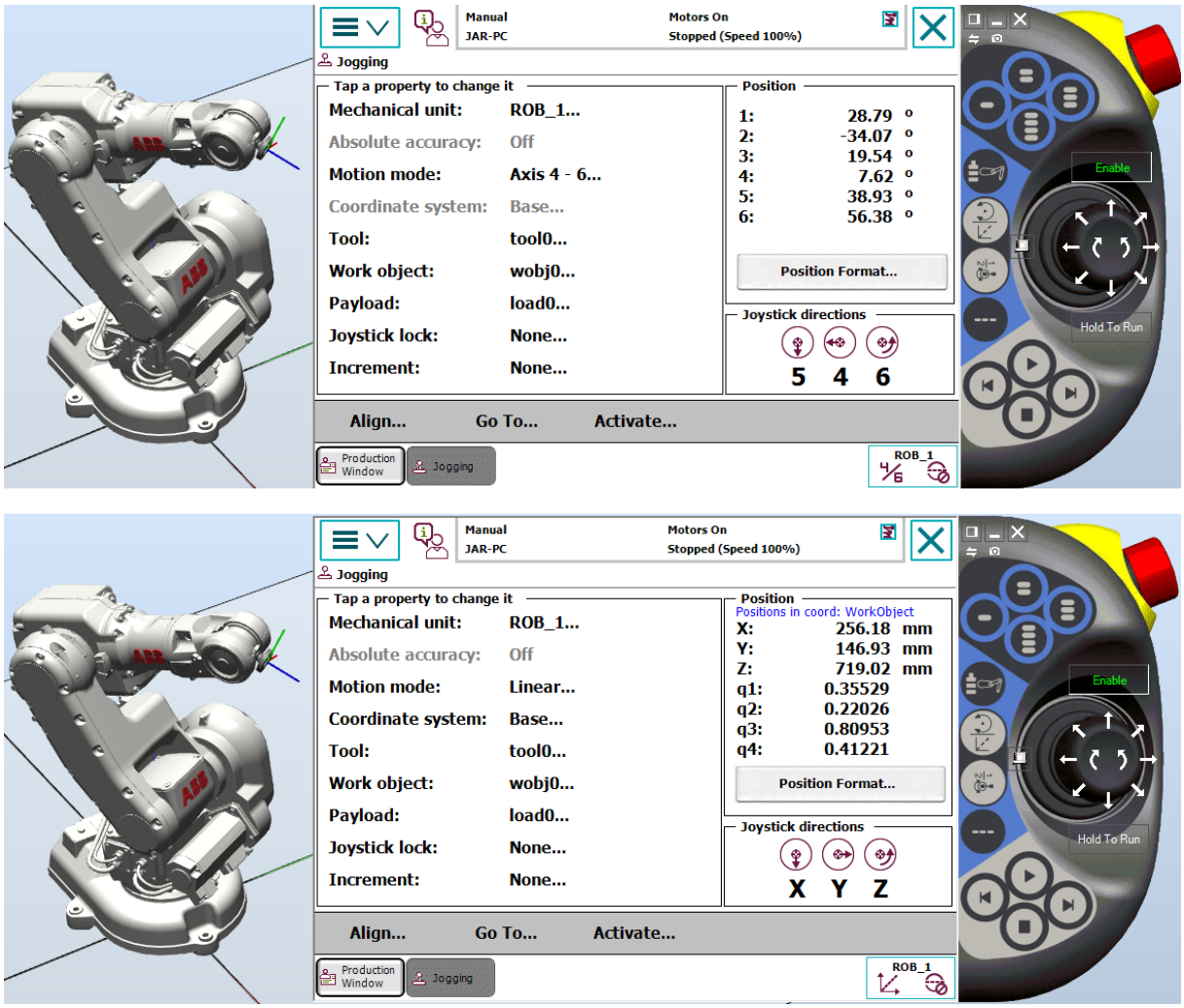



*Figure 3-34. Test no. 1.*

The resulting pose is:

```
p =

0.256169020979863   0.146951811198975   0.719041733432580
-0.355266700974361  -0.220219321995462  -0.809549214980808
-0.412212433003434
```

- Test no. 2

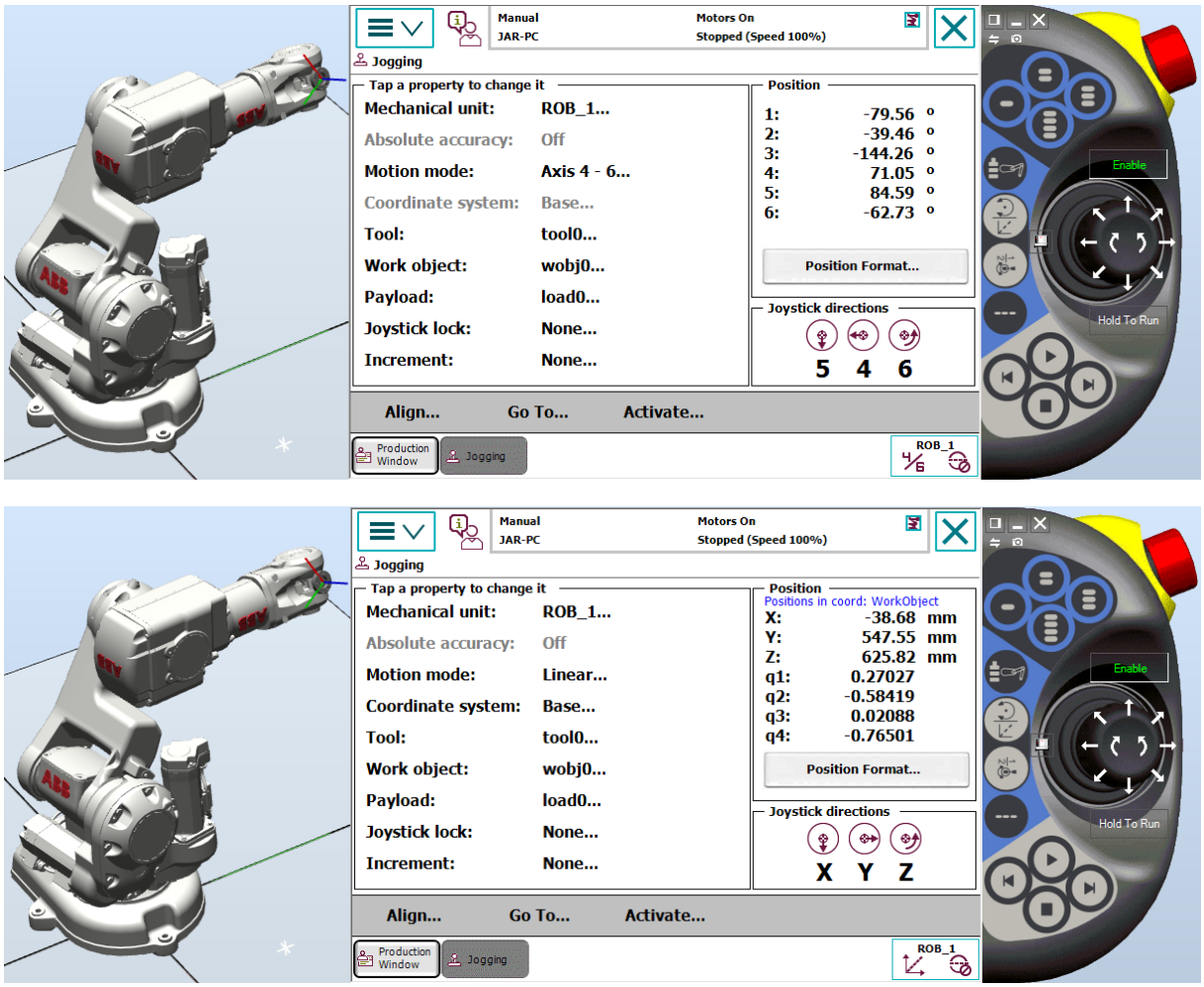



*Figure 3-35. Test no. 2.*

The resulting pose is:

```
p =

-0.038652641991810   0.547532574301628   0.625862378140840
-0.270294220765169   0.584219321953463  -0.020946045331934
0.764977176955064
```

- **Test no. 3**

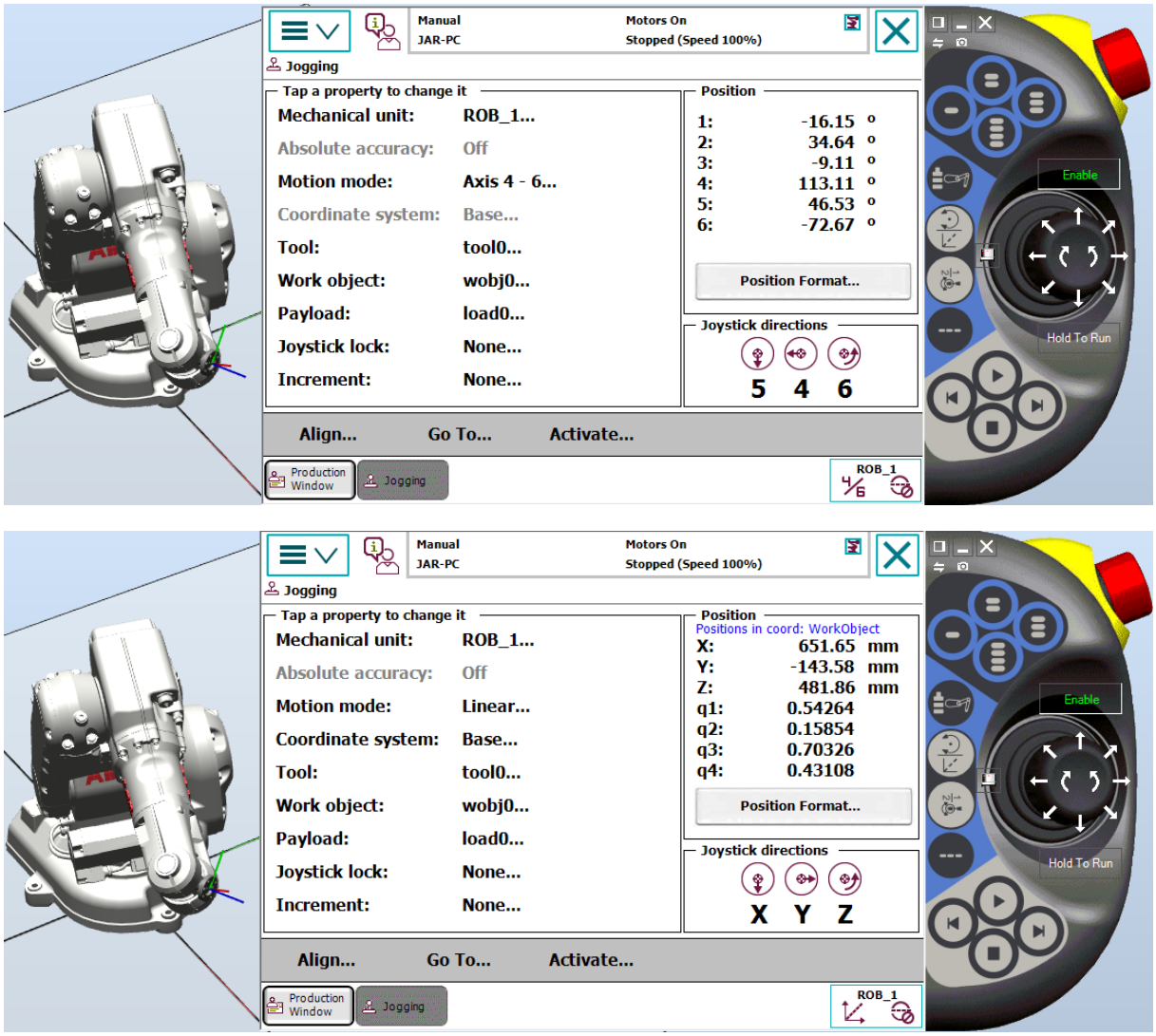



*Figure 3-36. Test no. 3.*

The resulting pose is:

```
p =

0.651650430407853  -0.143535605813641   0.481847014812847
-0.542604000206256  -0.158528784053503  -0.703280404944469
-0.431098823482308
```

All poses obtained come very close to the values calculated by RobotStudio up to the fifth decimal place in most cases. The model can be considered therefore valid. Note: The sign of some quaternions is switched, which make them still equivalent.

### 3.2.3 Newton-Euler algorithm

The dynamic model of the ABB IRB 140 has been wholly obtained through Code 12. Although the Newton-Euler algorithm achieves a higher performance than other methods, it has the inconvenience of not generating clearly defined matrices such as $M(q)$, $C(q,\dot{q})$ and $G(q)$, these must be extracted separately. This is accomplished with the MATLAB function to convert linear equation systems into their matrix form `equationsToMatrix`.

The size of the matrices is large enough not to fit on the display screen and the use of the function `diary` is necessary. It exports the matrices to a text file to subsequently copy and paste them to the Computed Torque Controller function (Code 13)

### 3.2.4 Verification of the Newton-Euler algorithm

In order to test the validity of the code, a simpler 2-link mechanism was developed, following the example 11.2 of [10]. (Figure 3-37)

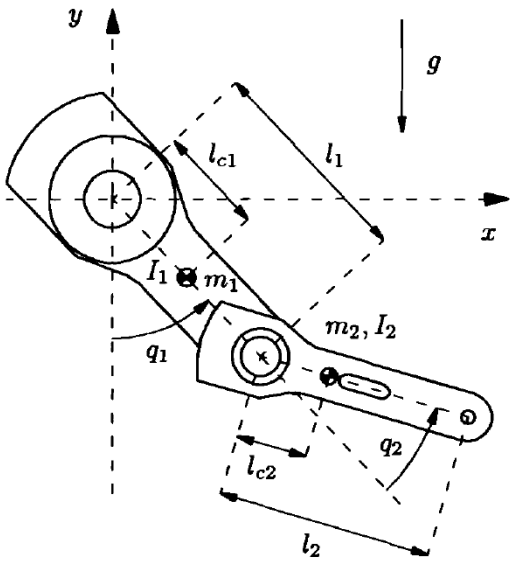



*Figure 3-37. 2 d.o.f. robot for the verification of the N-E algorithm. [10]*

Following the same steps as in 3.2.2, a simple 2-link robot was modelled on SimScape (Figure 3-38). All its properties can be found in Code 15.

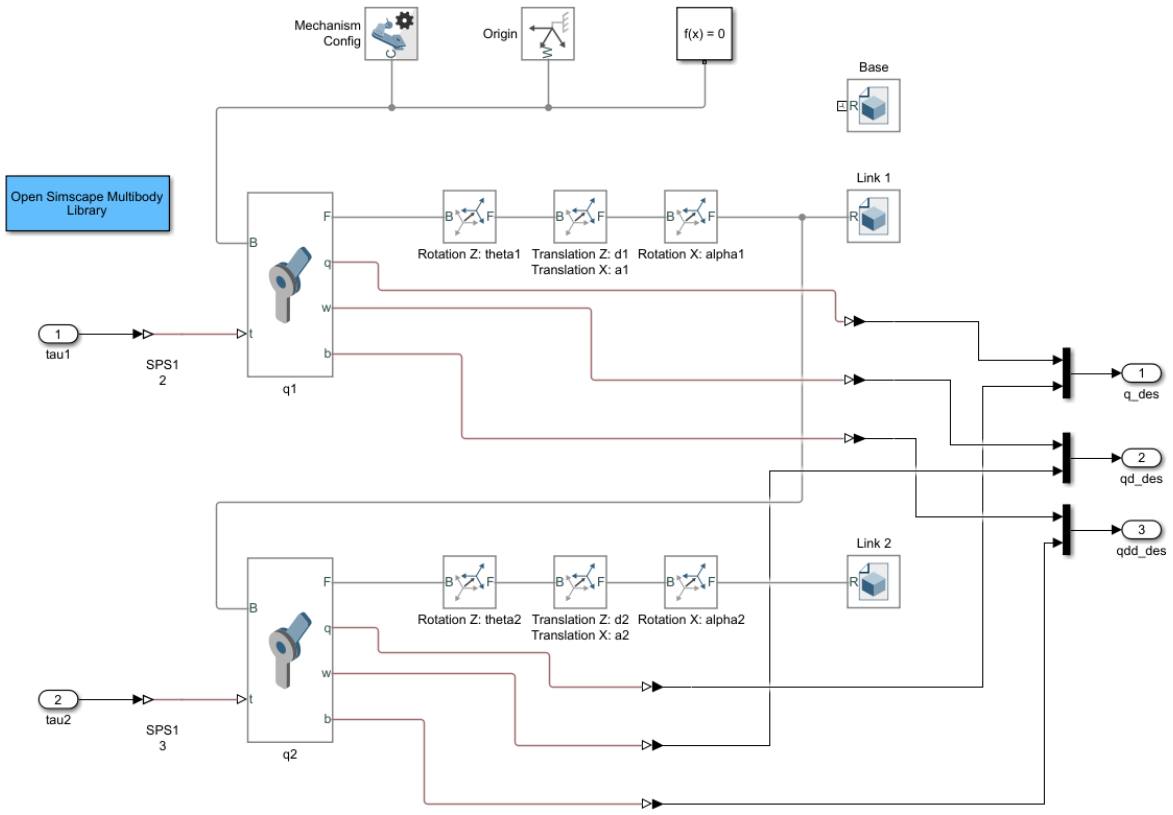



*Figure 3-38. 2-link robot SimScape schematic.*

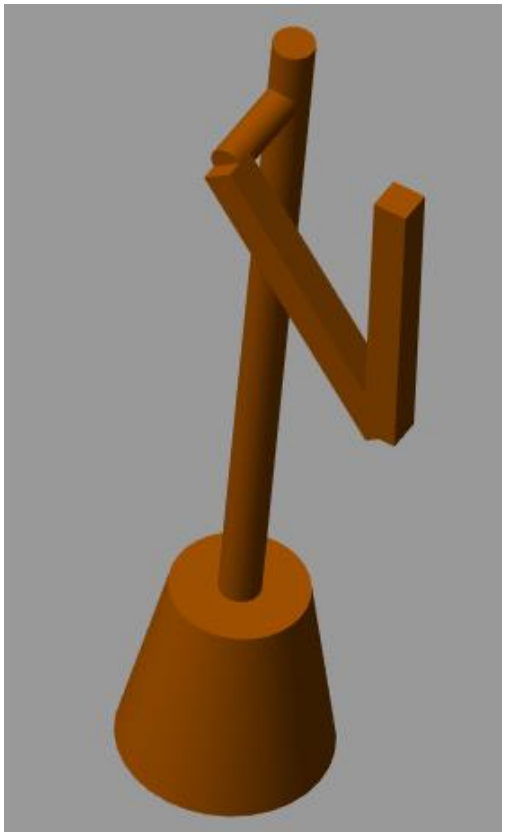


*Figure 3-39. 3-D representation of the 2-link robot for testing.*

This SimScape model is placed inside a control loop (Figure 3-40) that includes a Computed Torque Controller (Code 16) to which the matrices of the dynamic model of the 2-link robot have been added.

A joint coordinates generator provides a generic joint trajectory to just serve as test.

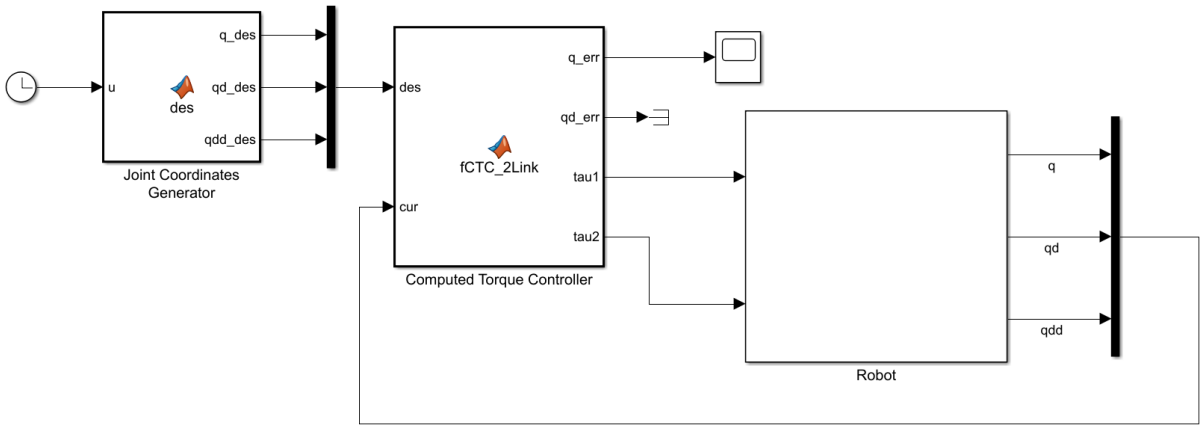
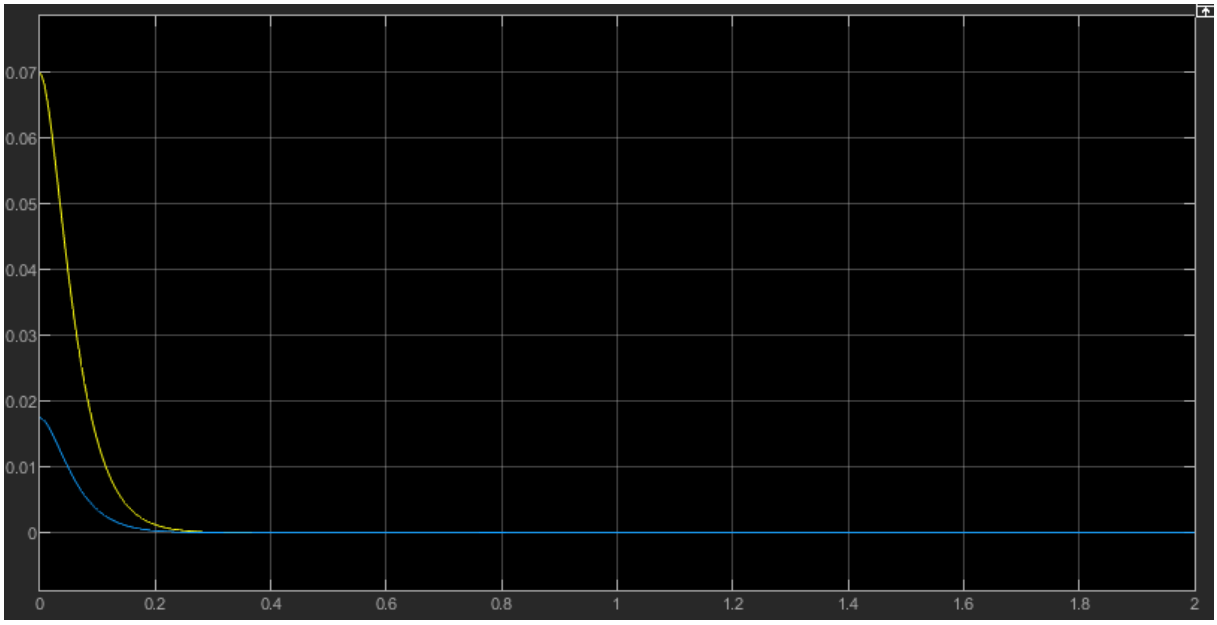


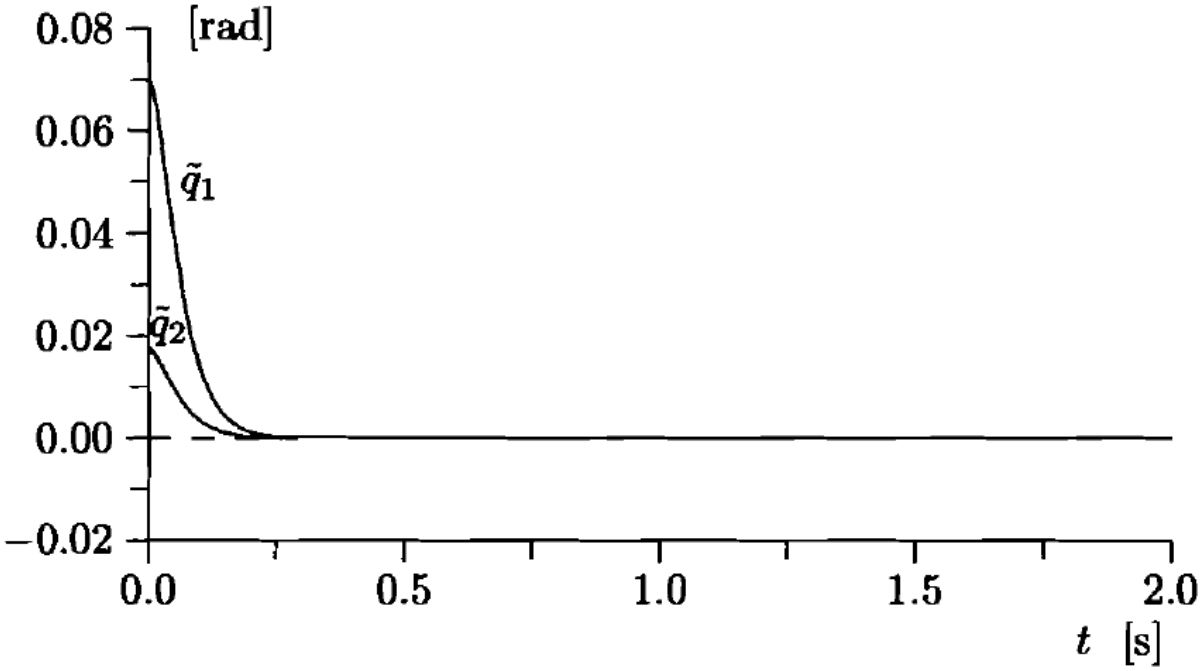

*Figure 3-40. Implementation of the 2-Link Robot CTC control loop.*

The resulting joint angle errors are:




*Plot 3-1. 2-link robot control loop. Joint angle errors in radians (yellow: $q_1$, blue: $q_2$).*

Results that match with those in [10]. Errors tend to zero due to the equivalence between the dynamic behavior of the SimScape model and the dynamic equations generated by Code 14.




*Plot 3-2. Joint angle errors present in Example 11.2 of [10].*

## 3.3 PID Controllers

Given that the input variables in the ABB IRB 140 are the joint torques, there can be only 6 PID mechanisms, one for each torque. This means only the 6 desired joint positions, the 6 desired joint velocities or the 6 desired joint accelerations can be used in the controller.

The high non-linearity of robots and complexity of the actuators could make the process of designing PIDs and the identification of the robot system parameters as complex as one would like to delve into it.

This section develops a series of PID controllers, tuned by the MATLAB tool PID Tuner under certain time constraints. The main objective is to obtain fairly good performances that could serve as comparison to the CTC Controller.

### 3.3.1 Initialization of PIDs and obtention of these initial torque values

Due to the initial robot configuration, the integral part of the controllers was initialized at the required torques to keep the robot in that initial configuration. The calculation of these necessary values is performed by modifying the joint blocks, so they receive an "angle signal" and throw the corresponding torque in response.

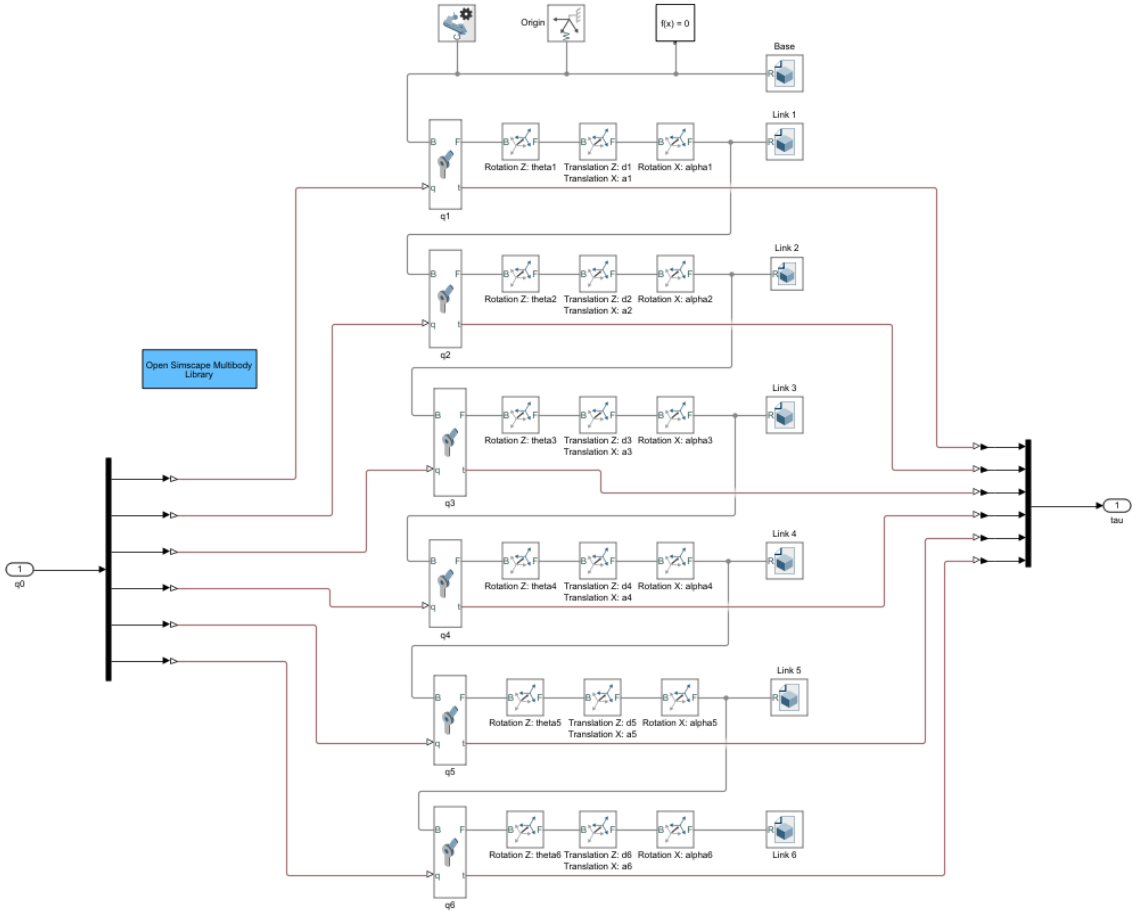



*Figure 3-41. Robot schematic for the calculation of the initial torques.*

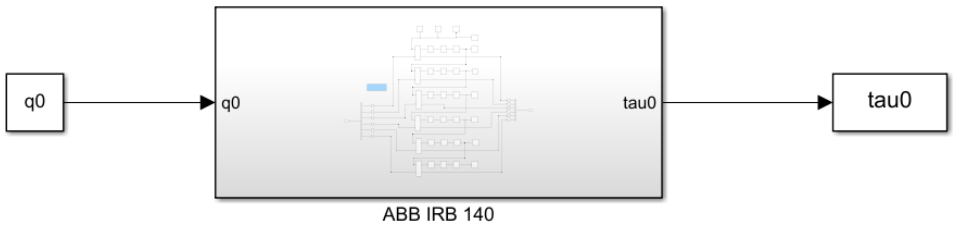



*Figure 3-42. Schematic for the calculation of initial torques.*

These torques are then extracted and copied on the initialization code for the PID loop.

### 3.3.2 PID tuning

Making now use of the PID Tuner app integrated on MATLAB, the software linearizes the robot model at home position (zero joint angles) and identifies its parameters to recommend $K_p$, $K_i$ and $K_d$ values for the PID controllers.

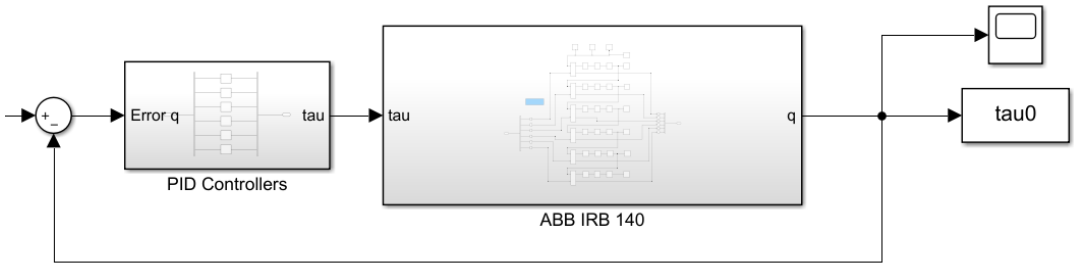



*Figure 3-43. Simplified closed-loop with PID controllers and Robot System.*

The $q$ vector is demuxed for the block to be able to feed each joint angle signal to its correspondent PID. The output vector of torques is then muxed again, acting this vector as output.

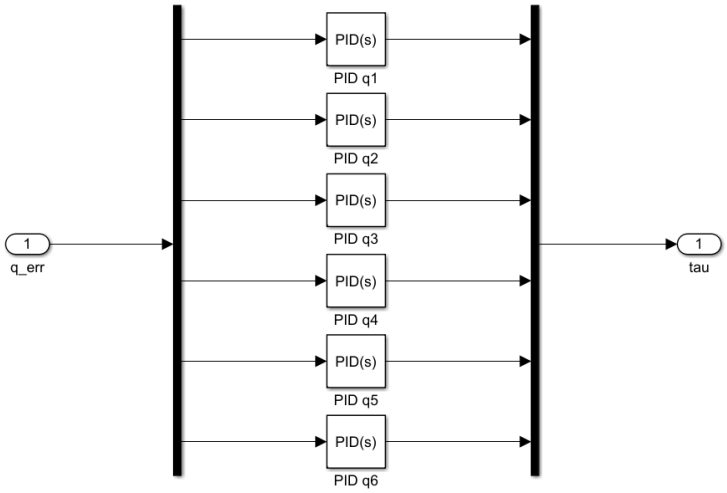



*Figure 3-44. PID battery block content*

After placing the robot system block inside a closed loop with the controllers, simply by clicking *Tune* in the PID block properties (Figure 3-45), it automatically calculates and offers a set of parameters for the controller (Figure 3-46).

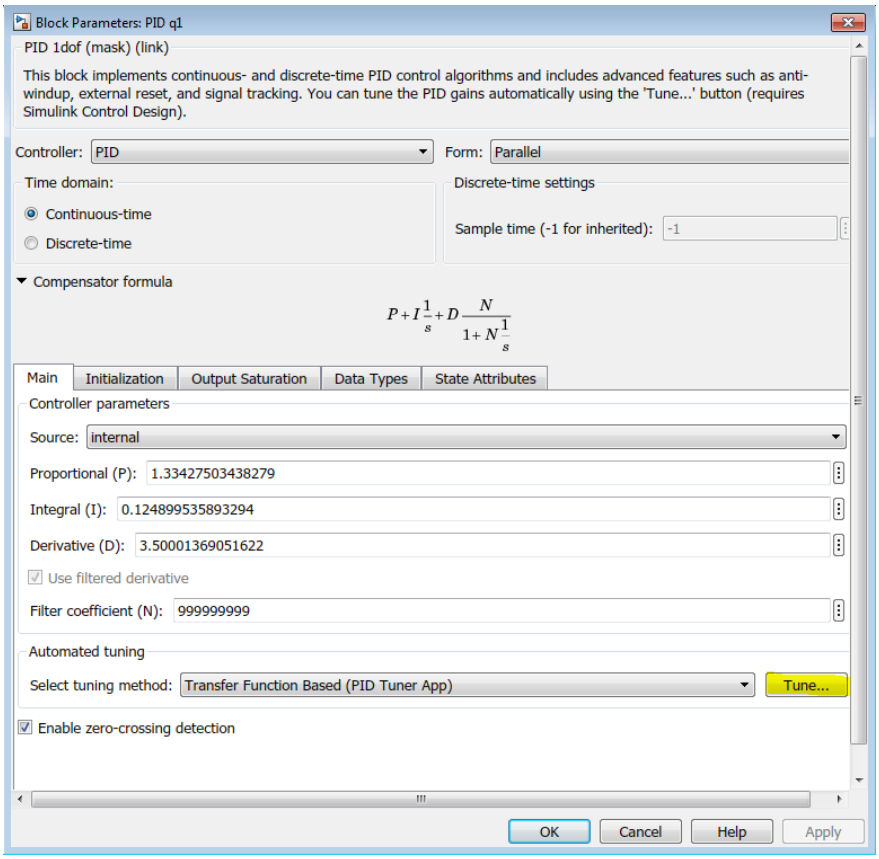



*Figure 3-45. PID block parameters.*

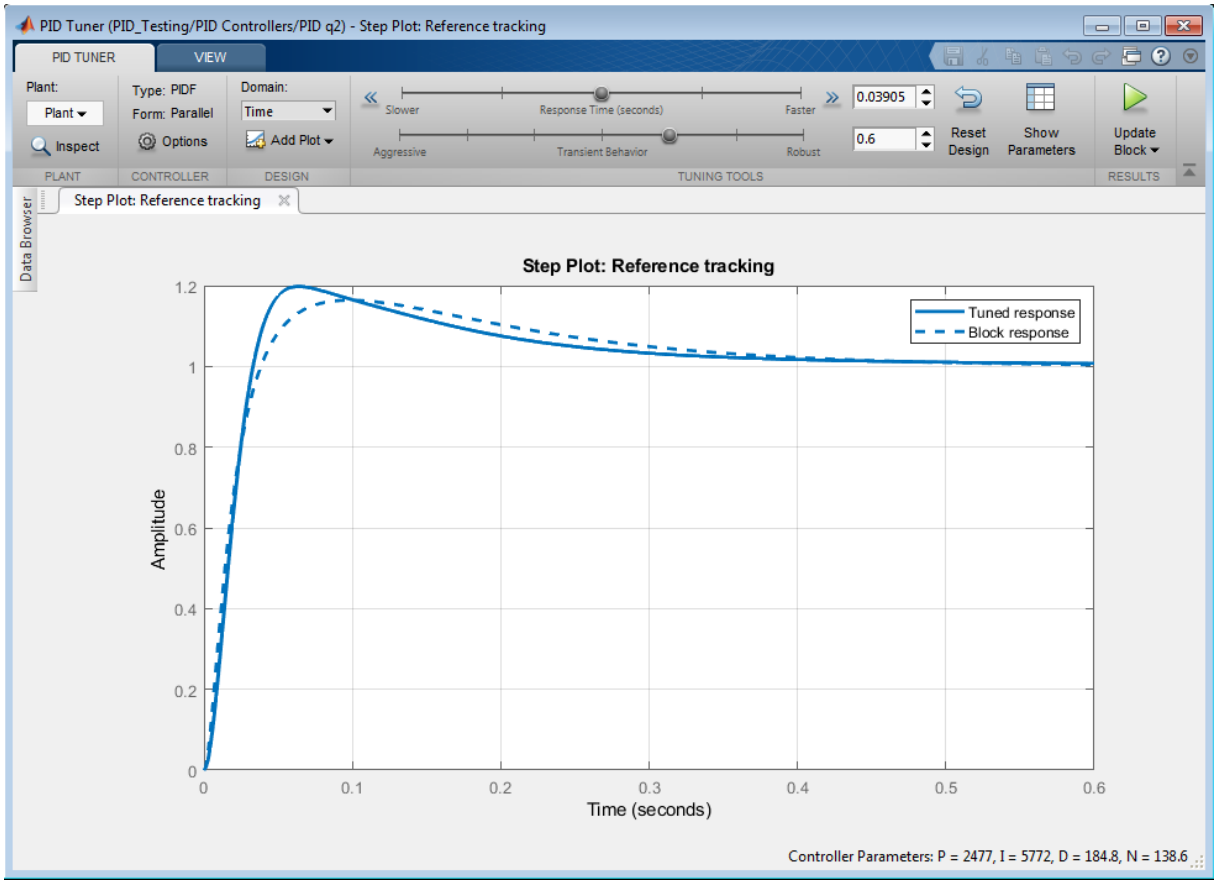



*Figure 3-46. Example window of the PID Tuner.*

### 3.3.3 Final PID values

An aggressive PID design was attempted, setting the response time for each closed-loop system to 0.01 seconds. The resulting PID parameters are shown in the following Table 3-7.

| PID | $K_p$ | $K_i$ | $K_d$ | $N$ |
|---|---|---|---|---|
| 1 | 13875.5679812054 | 136120.974518283 | 347.129456680199 | 1090.79405084324 |
| 2 | 35554.9091580814 | 399521.426748228 | 745.761367338793 | 729.364813354201 |
| 3 | 4941.70226164605 | 51928.4446602287 | 113.296542117102 | 729.364813354201 |
| 4 | 55.9972754306752 | 527.478956238882 | 1.45998670887717 | 1090.79405084324 |
| 5 | 3.04185650253597 | 29.2717188604941 | 0.0776376501943788 | 1090.79405084324 |
| 6 | 0.0949261549260263 | 0.895070977633301 | 0.00247254567688776 | 1090.79405084324 |

*Table 3-7. PID parameters.*

## 3.4 Computed Torque Controller

A MATLAB function block was programmed, containing this control method. For the sake of simplicity, the block receives a common value of bandwidth $\omega$ for all joint actuators from Workspace, plus the desired joint coordinates, their derivatives and the current joint coordinates and their derivatives.

The current and desired values of joint angles, velocities and accelerations are then substituted in the vast symbolic expressions of matrices $M$, $C$ and $G$ to obtain the numeric versions of these matrices. Then the control law is stated in order to extract the desired torques to exert in each joint.

$$Q = M(q^{\#})\left[\ddot{q}^{*} + K_v(\dot{q}^{*} - \dot{q}^{\#}) + K_p(q^{*} - q^{\#})\right] + C(q^{\#}, \dot{q}^{\#})\dot{q}^{\#} + G(q^{\#}) \qquad (3.6)$$

The implemented function can be found in Code 12.

## 3.5 Full system implementation and results

Following the schematic proposed in Figure 1-5, all blocks have been positioned, forming the schematic depicted in Figure 3-47.

The selected testing trajectory is the following: Starting from a home position where all joint angles are zero, the robot should move to a point in front of it (same $y$-axis value), draw a circle inside the $xy$ plane starting and finishing at the same point. Then return to home position and wait 1 second. Repeat this process twice with different circle radii.

| Movement | Description |
|---|---|
| **1** | From Home to Initial Point of Circle 1 (small) |
| **2** | Tracing of Circle 1 |
| **3** | From Initial Point of Circle 1 to Home |
| **4** | Wait 1 second |
| **5** | From Home to Initial Point of Circle 2 (medium) |
| **6** | Tracing of Circle 2 |
| **7** | From Initial Point of Circle 2 to Home |
| **8** | Wait 1 second |
| **9** | From Home to Initial Point of Circle 3 (large) |
| **10** | Tracing of Circle 3 |
| **11** | From Initial Point of Circle 3 to Home |

*Table 3-8. Testing trajectory broken down in movements.*

A new block was added that finds the pose errors produced, in other words, the difference between the setpoints and the actual pose reached by the robot in the simulation. This MATLAB function block corresponds to Code 17.

After simulation, many key variables will be sent to Workspace and saved as .mat files. Operating with some of them through a script, the goal is to obtain some parameters of interest such as pose errors and input torques.

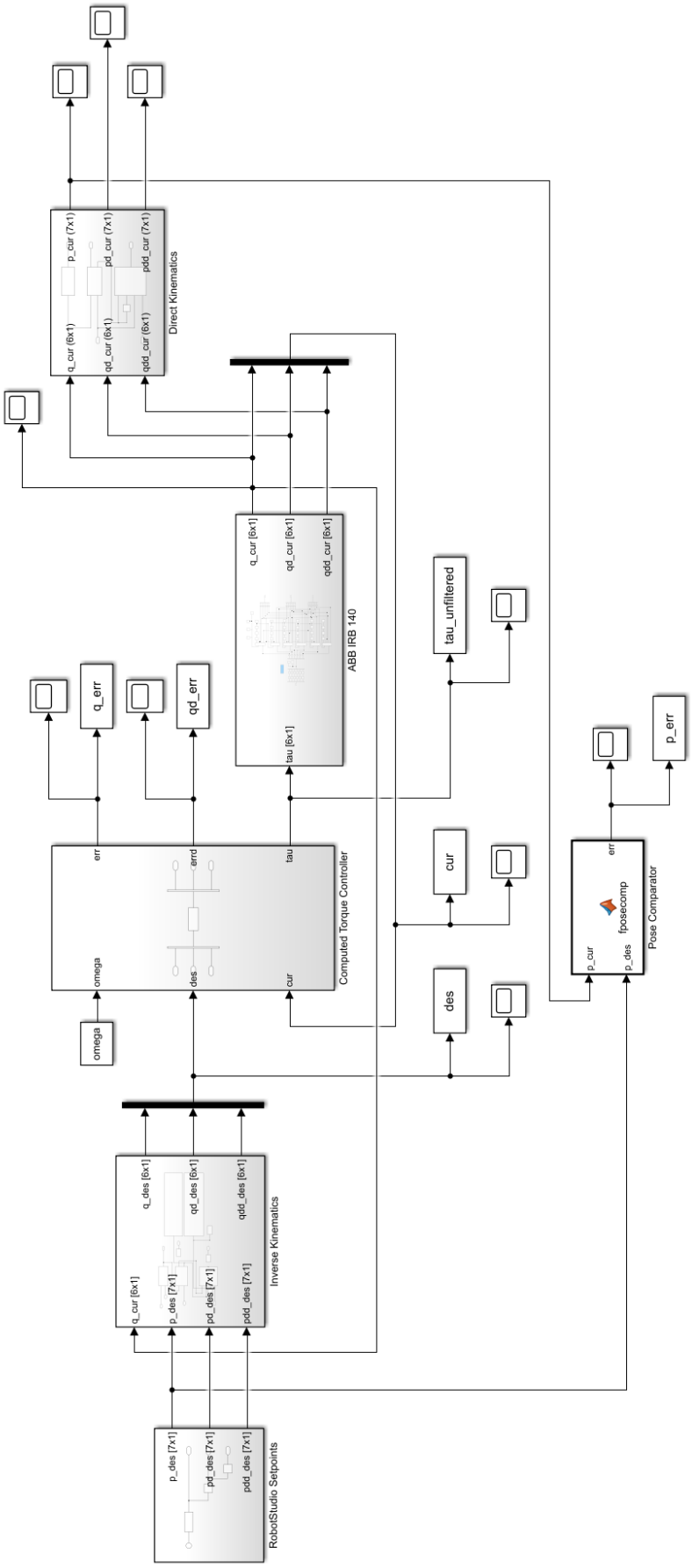



*Figure 3-47. Full system schematic for a Computed Torque Controller under RobotStudio data.*

### 3.5.1 Full system model with RobotStudio data as input and a Computed Torque Controller

After first contact with the software, building a time-dependent setpoint signal that could be replicated later on MATLAB did not seem a trivial task. For this reason, it was decided to build the testing trajectory on RobotStudio without being bounded to time requirements. The results would be extracted and used as setpoints for the MATLAB model, to see how the results of the latter can approach to the firsts.

All positions and orientations forming pose vectors were represented on RobotStudio and rearranged to shape the desired testing trajectory. The result was converted into RAPID code in Code 19 and is shown graphically in Figure 3-48. Speed of the end effector was set at v300 ($300\ [mm/s]$) and precision at fine (the best the robot can achieve).

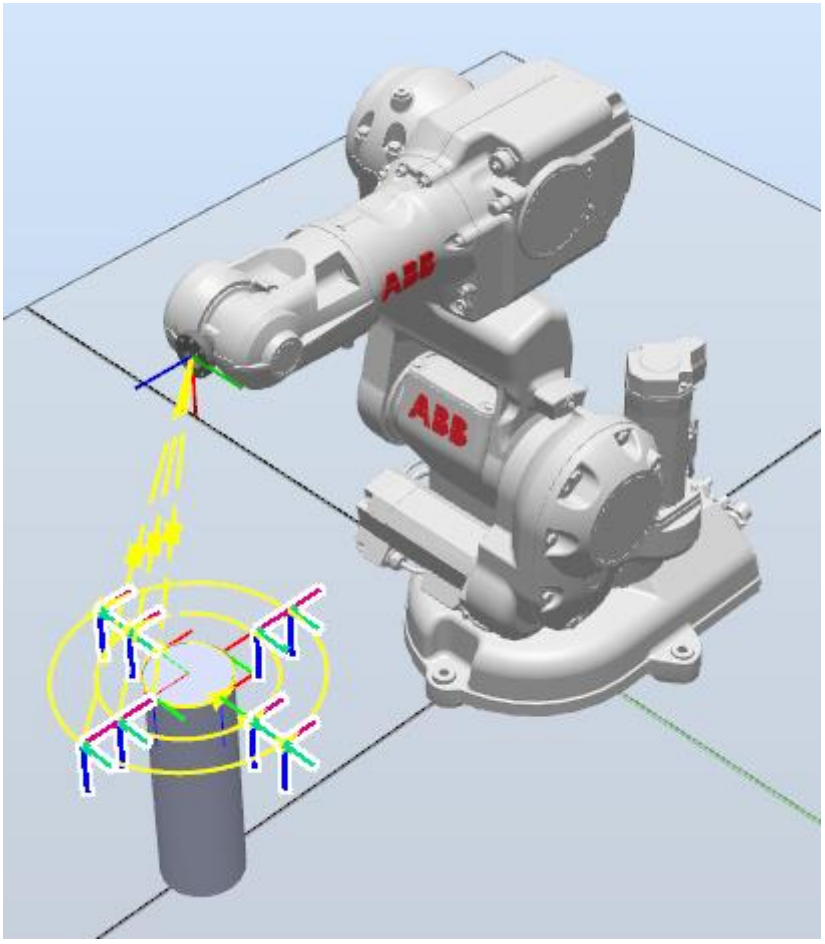



*Figure 3-48. Spatial representation of the testing trajectory on RobotStudio.*

Before running the code, simulation options were optimized to get more accurate results with a lower timestep which would result in more samples. (Figure 3-49). Unluckily, the lowest sampling time available is not guaranteed and higher than desired, which leads to imprecise depiction of the real motion of the robot. In addition, RobotStudio limits the information that can be extracted from simulations. For example, exerted torques of any kind or individual joint powers are not an option in the list of extractable signals. For this reason, the final set of relevant signals extracted from simulation was:

- Pose vectors (position and orientation)
- Joint angles
- Total power

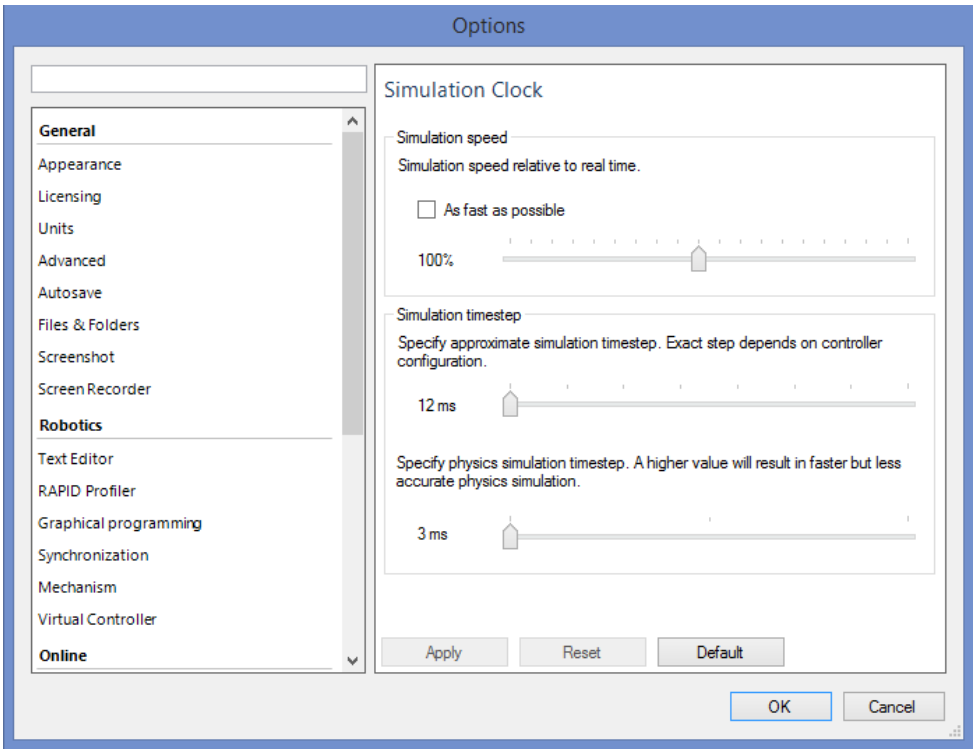



*Figure 3-49. RobotStudio simulation configuration menu.*

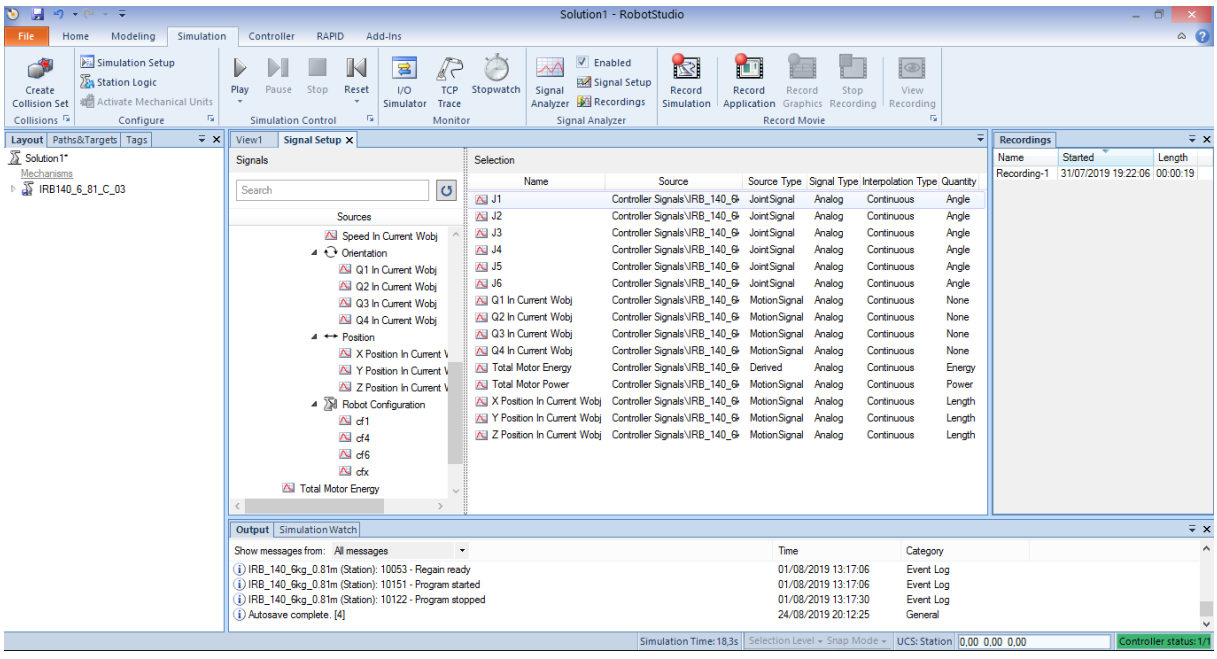



*Figure 3-50. RobotStudio signal setup menu.*

The signals were then stored in an Excel file and later converted into a .mat file, the native extension developed by MATLAB to store variables. The file was named RobotStudio_rawdata.mat and saved in the same folder as the main scripts.

Since the collected data are discretized and only count with the pose vectors (Table 3-9), the first issue comes with the calculation of accelerations. A double derivative of discretized positions along short time steps results in extremely abrupt acceleration values that are not accurate.

| Time | y | z | Total Power | q1 | q2 | q3 | q4 | x | Energy |
|---|---|---|---|---|---|---|---|---|---|
| 0,000 | 0,000 | 712,000 | 2,395 | 0,707 | 0,000 | 0,707 | 0,000 | 515,000 | 0,000 |
| 0,048 | 0,000 | 712,000 | 2,395 | 0,707 | 0,000 | 0,707 | 0,000 | 515,000 | 0,000 |
| 0,072 | 0,000 | 712,000 | 2,395 | 0,707 | 0,000 | 0,707 | 0,000 | 515,000 | 0,000 |
| 0,096 | 0,000 | 709,598 | 15,463 | 0,704 | 0,000 | 0,711 | 0,000 | 516,707 | 1,484 |
| 0,120 | 0,000 | 704,019 | 35,196 | 0,695 | 0,000 | 0,719 | 0,000 | 520,595 | 2,329 |
| 0,144 | 0,000 | 698,010 | 32,255 | 0,686 | 0,000 | 0,727 | 0,000 | 524,665 | 3,103 |
| 0,168 | 0,000 | 691,944 | 31,261 | 0,677 | 0,000 | 0,736 | 0,000 | 528,650 | 3,854 |
| 0,192 | 0,000 | 685,824 | 28,149 | 0,668 | 0,000 | 0,744 | 0,000 | 532,551 | 4,529 |
| 0,216 | 0,000 | 679,650 | 27,578 | 0,659 | 0,000 | 0,752 | 0,000 | 536,367 | 5,191 |
| 0,240 | 0,000 | 673,424 | 27,050 | 0,649 | 0,000 | 0,760 | 0,000 | 540,097 | 5,840 |
| 0,264 | 0,000 | 667,147 | 26,492 | 0,640 | 0,000 | 0,768 | 0,000 | 543,740 | 6,476 |
| 0,288 | 0,000 | 660,819 | 25,943 | 0,630 | 0,000 | 0,776 | 0,000 | 547,296 | 7,099 |
| 0,312 | 0,000 | 654,443 | 25,408 | 0,621 | 0,000 | 0,784 | 0,000 | 550,763 | 7,708 |
| 0,336 | 0,000 | 648,019 | 24,855 | 0,611 | 0,000 | 0,792 | 0,000 | 554,141 | 8,305 |
| 0,360 | 0,000 | 641,549 | 24,309 | 0,601 | 0,000 | 0,800 | 0,000 | 557,429 | 8,888 |
| 0,384 | 0,000 | 635,034 | 23,760 | 0,591 | 0,000 | 0,807 | 0,000 | 560,627 | 9,459 |
| 0,408 | 0,000 | 628,475 | 23,240 | 0,580 | 0,000 | 0,814 | 0,000 | 563,734 | 10,016 |
| 0,432 | 0,000 | 621,873 | 22,716 | 0,570 | 0,000 | 0,822 | 0,000 | 566,749 | 10,562 |
| 0,456 | 0,000 | 615,229 | 22,189 | 0,560 | 0,000 | 0,829 | 0,000 | 569,672 | 11,094 |
| 0,480 | 0,000 | 608,545 | 21,659 | 0,549 | 0,000 | 0,836 | 0,000 | 572,502 | 11,614 |
| 0,504 | 0,000 | 601,823 | 21,131 | 0,538 | 0,000 | 0,843 | 0,000 | 575,238 | 12,121 |

*Table 3-9. RobotStudio raw data extract.*

The proposed solution to overcome this inconvenient was to interpolate all 7 pose components using the Curve Fitting Tool, another app present in MATLAB. It allows to interpolate different vectors through numerous interpolating algorithms. The algorithm selected for this specific problem was the Shape-preserving (PCHIP) interpolant, resulting in nice fitting approximations all along the simulation results.

In order to implement these interpolations on Simulink, an Interpreted MATLAB Function was declared, calling the function present in Code 20. This function calls the interpolated data in the form of cfit and substitutes values in them.

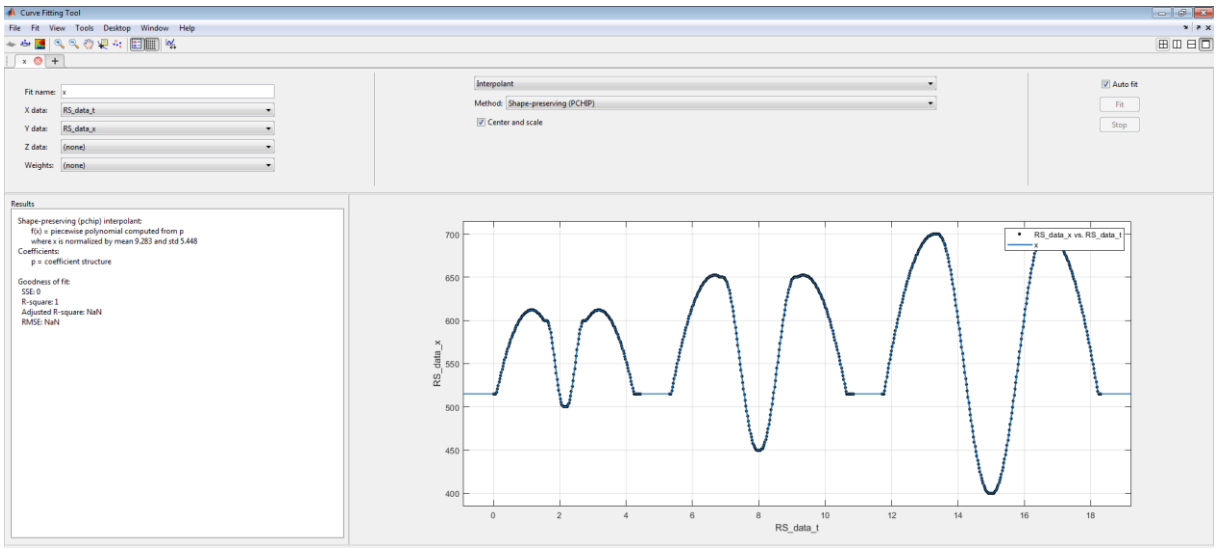
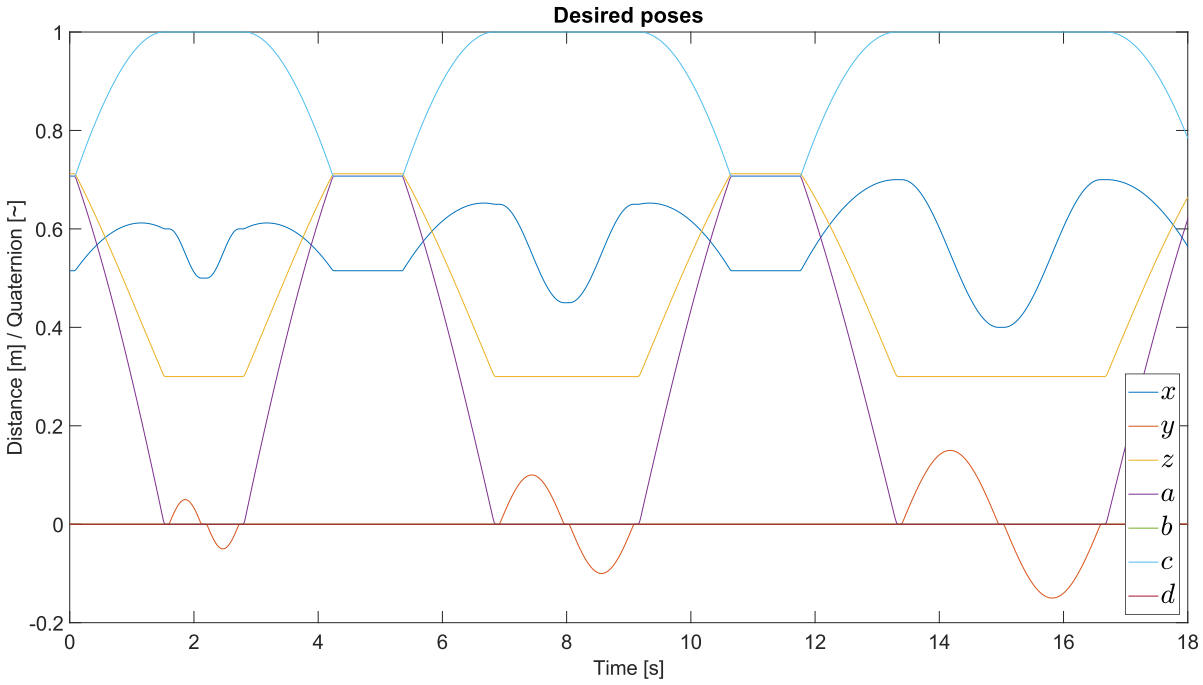
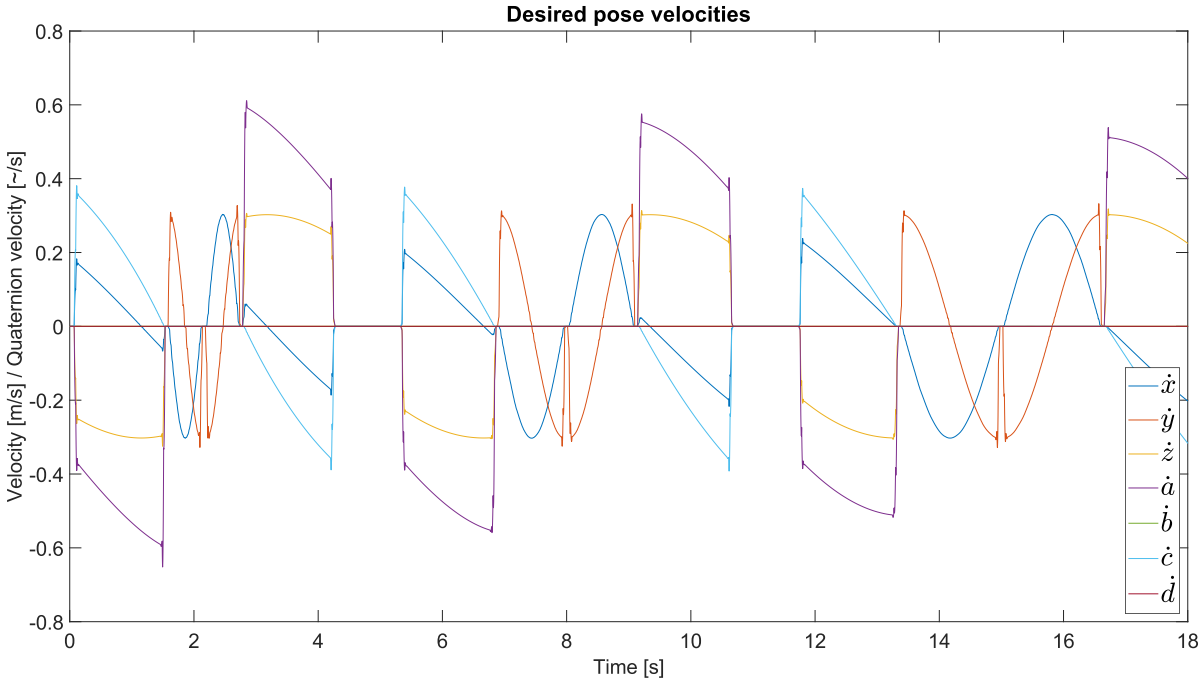



*Figure 3-51. Interpolation of x-axis results of the RobotStudio simulation using the PCHIP algorithm.*

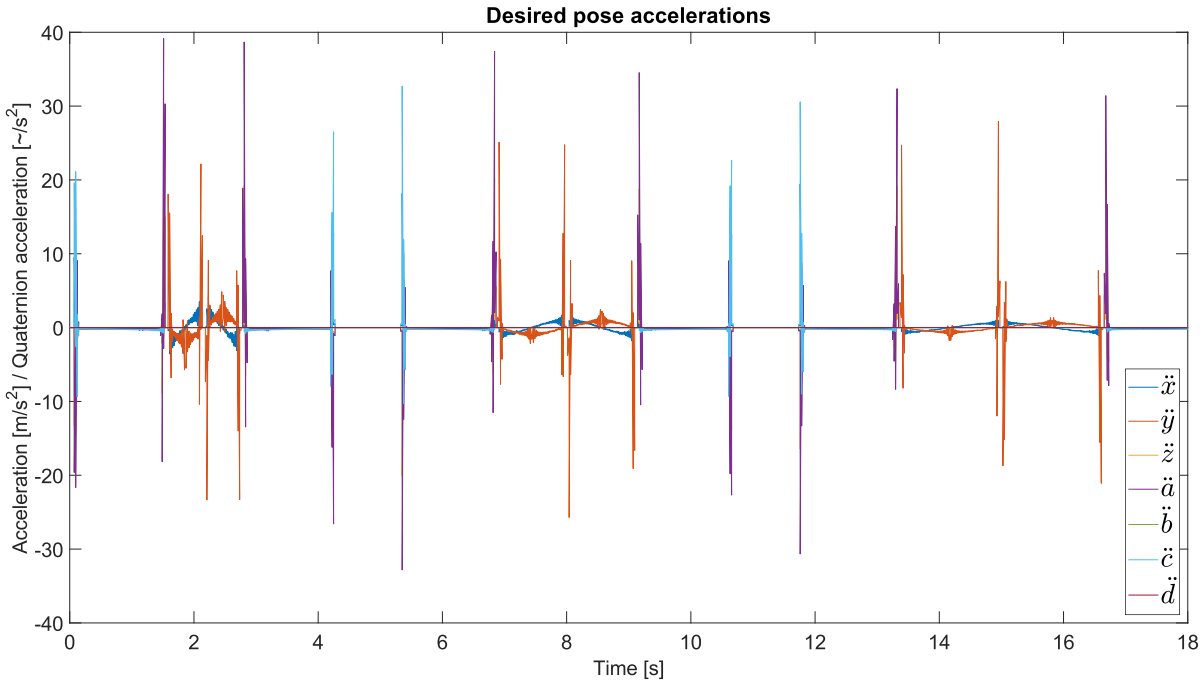
The resulting desired poses and its derivatives are the following:




*Plot 3-3. Desired interpolated poses extracted from RobotStudio data.*



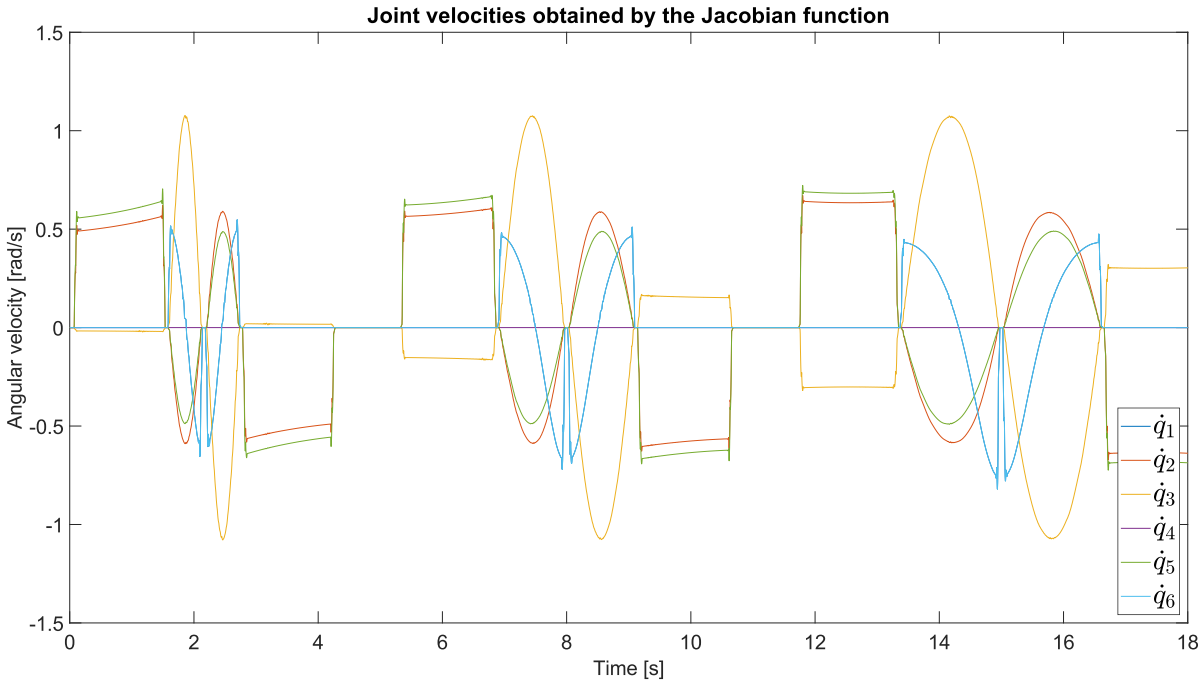

*Plot 3-4. Pose velocities extracted from RobotStudio data.*



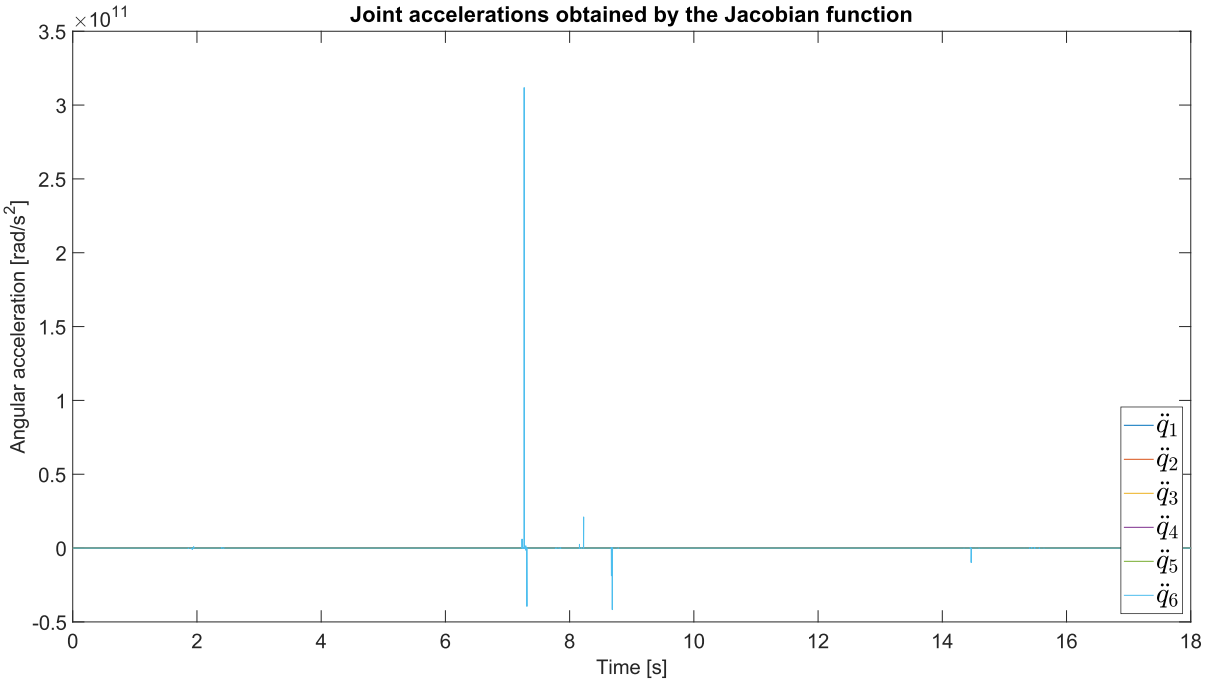

*Plot 3-5. Pose accelerations extracted from RobotStudio data.*

Velocities and accelerations were obtained by placing Simulink derivative blocks at the end of this Interpreted MATLAB Function block. The result while running inside the full schematic of Figure 3-47 is far from what was expected. Derivative blocks result highly problematic with very short time steps. Due to the mandatory condition by the SimScape Multibody robot system of using a variable-step solver, several bursts in the velocity and acceleration signals are being generated at the beginning of the simulation, generating subsequently in the controller large torque signals.

Calculating in advance the pose derivatives and storing them in vectors was not successful, nor interpolating these preconstructed vectors and introducing them in a function. Although the Jacobian function for velocities works as expected (Plot 3-6), the derivative of this function does not. Sudden extreme bursts appear and make the problem unapproachable (Plot 3-7). The most plausible reason is the slight inaccuracies of the interpolated discrete data, that creates unwanted effects inside the Jacobian that leads the function to lose control.




*Plot 3-6. Joint velocities obtained by the Jacobian function with RobotStudio data.*




*Plot 3-7. Joint accelerations obtained by the Jacobian function with RobotStudio data.*

Due to these results, a new way to find the derivative of the joint angles signal was attempted. This way consists of deriving the pose signal directly by the means of Simulink derivative blocks, as shown in Figure 3-52.

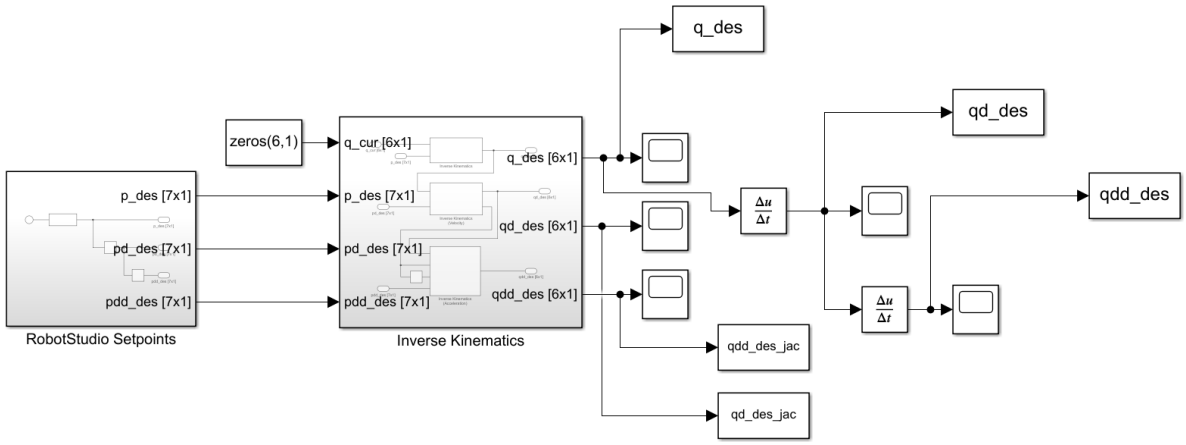
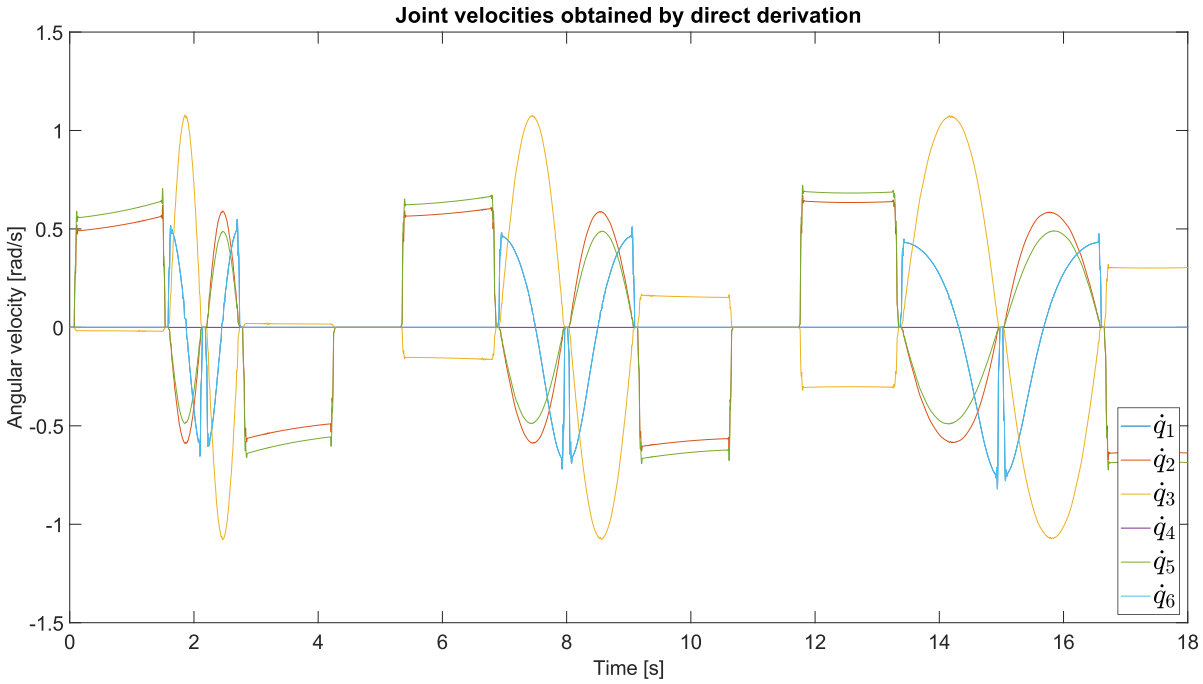
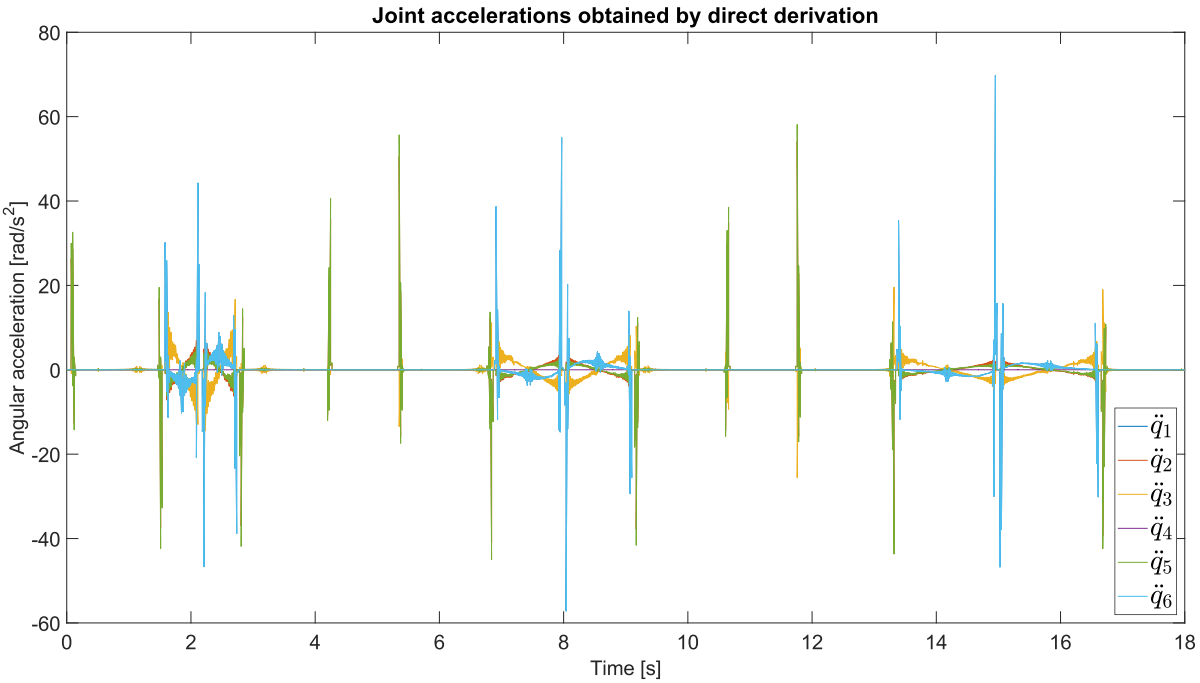



*Figure 3-52. Precalculation of joint angles* $\mathbf{q}$ *and their derivatives.*

Joint velocities and accelerations take now the following form:




*Plot 3-8. Joint velocities obtained by direct derivation with RobotStudio data.*




*Plot 3-9. Joint accelerations obtained by direct derivation with RobotStudio data.*

Although the sudden bursts in the acceleration signals have been fixed, they still appear to oscillate. Regardless of this, joint velocities and accelerations, along with the joint angles are placed at the beginning of the control system (Figure 3-53) to be further tested. Resulting torques for each joint and pose errors of the end effector are presented in the next sections.

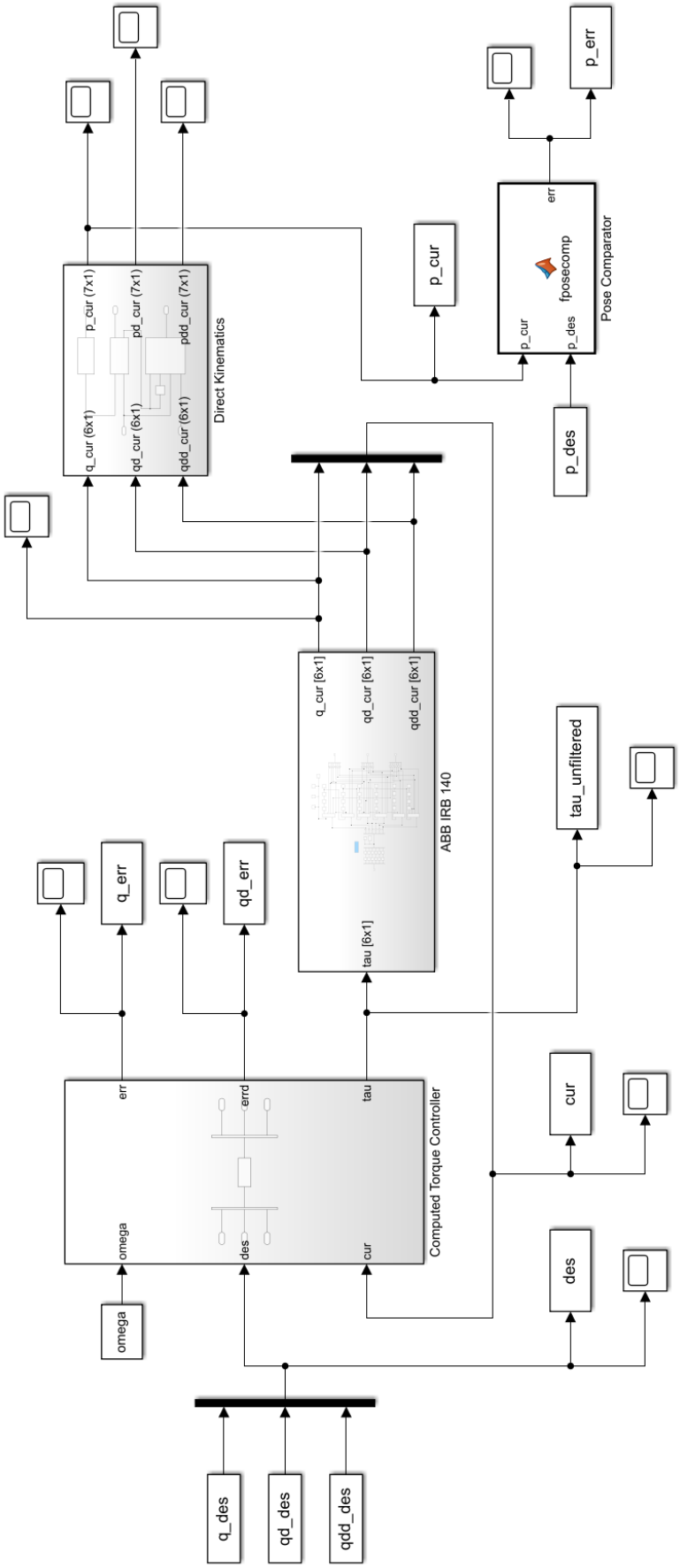
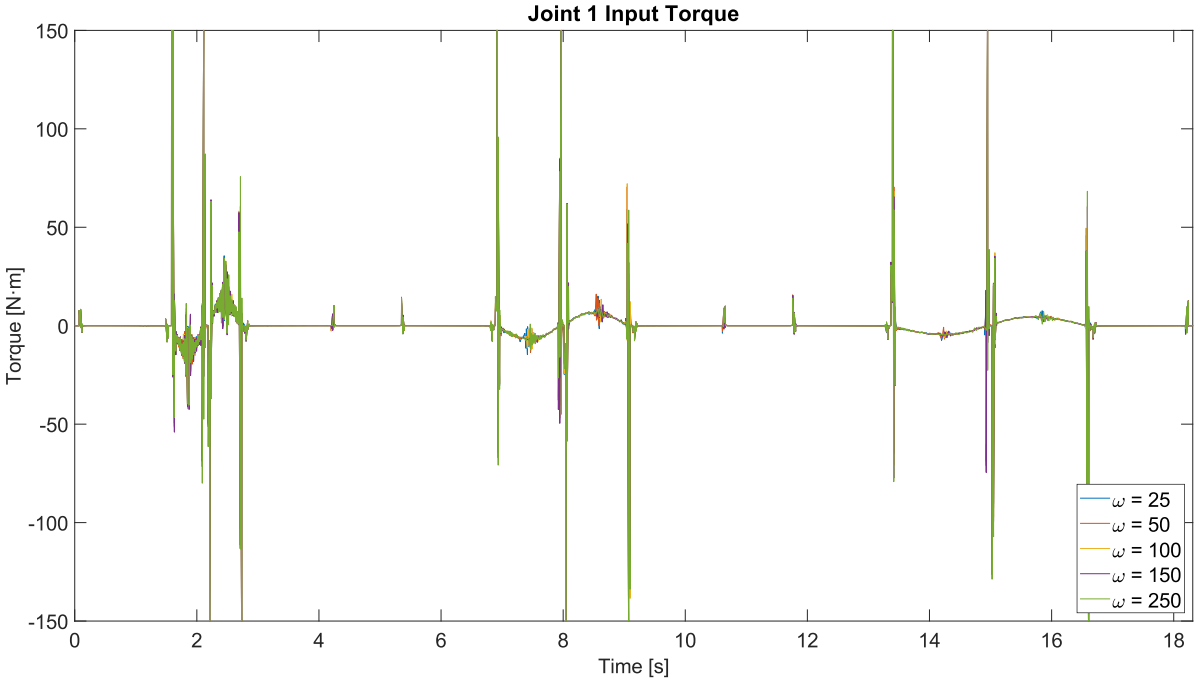



*Figure 3-53. Reduced system schematic for a CTC with precalculated joint coordinates.*

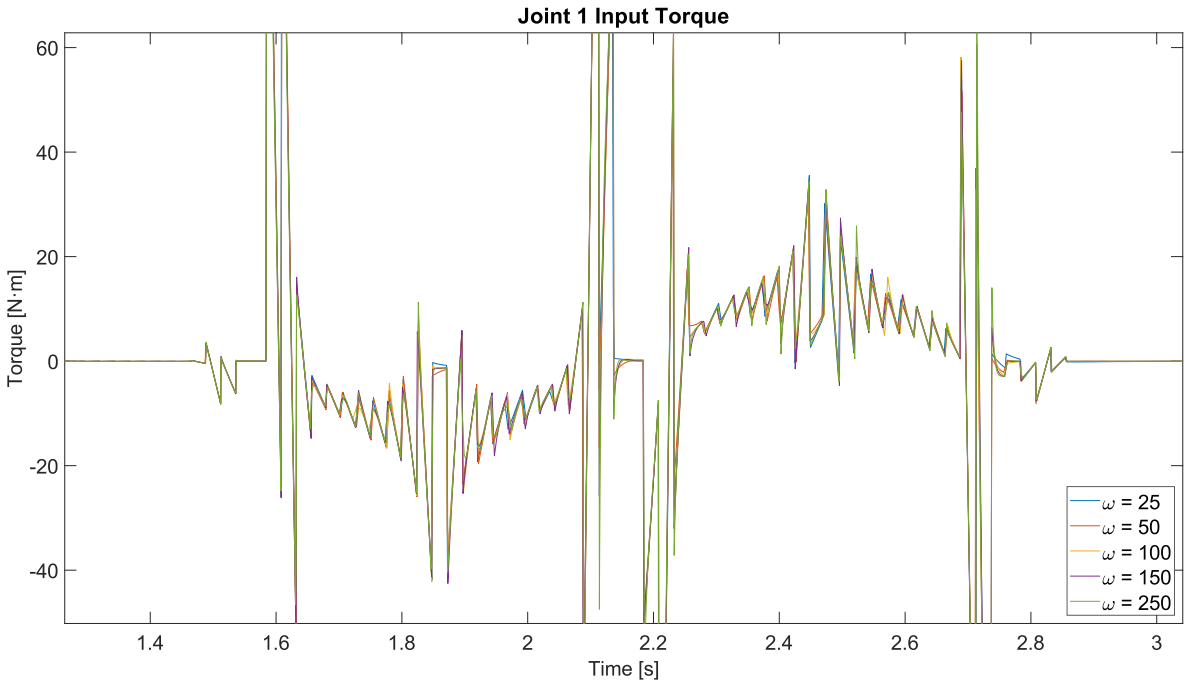
#### *3.5.1.1 Input torques*

$\omega$ represents the common design bandwidth of the Computed Torque Controller, as stated in section 3.4. Every plot depicts the torques generated in a certain joint by the CTCs with different values of $\omega$.

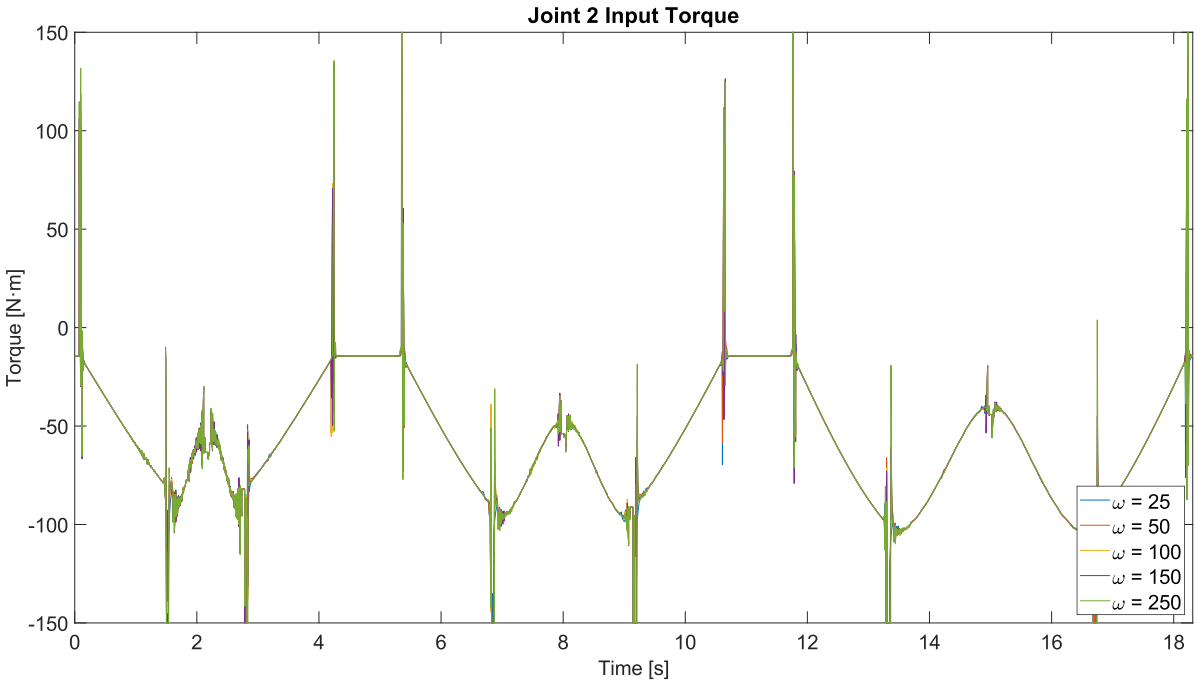



*Plot 3-10. CTC Controller with RobotStudio trajectory. Input torque in joint 1.*

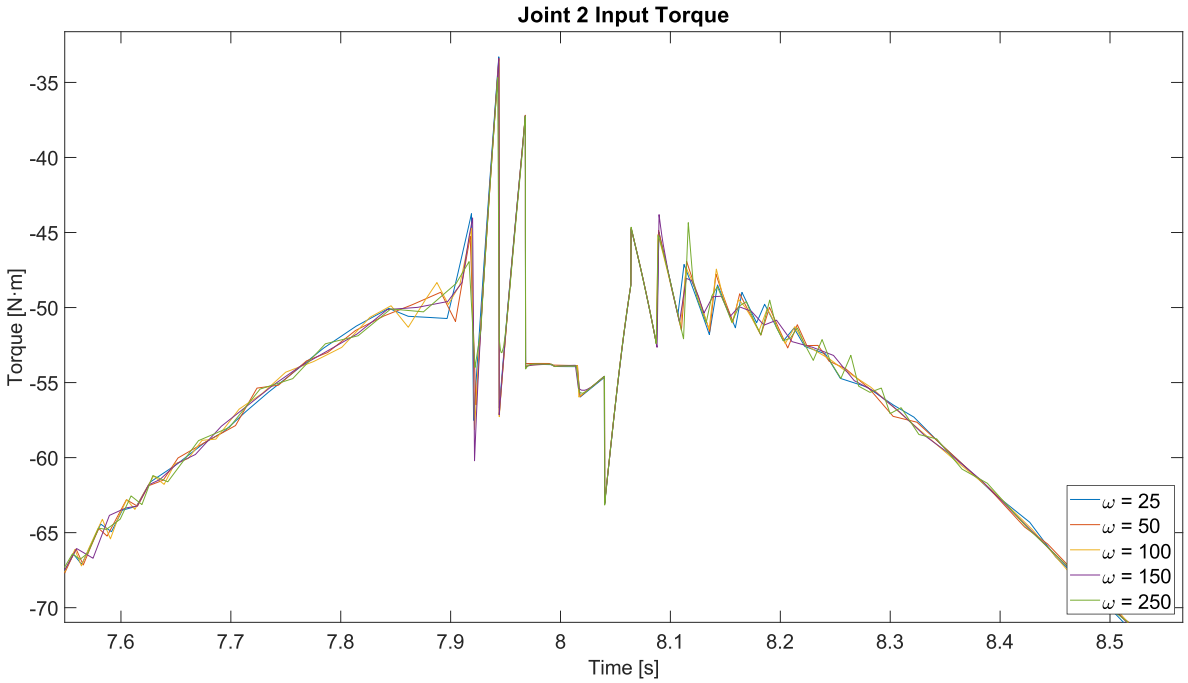
Due to the proximity of all signals, a close-up of some areas of the plots will be provided.



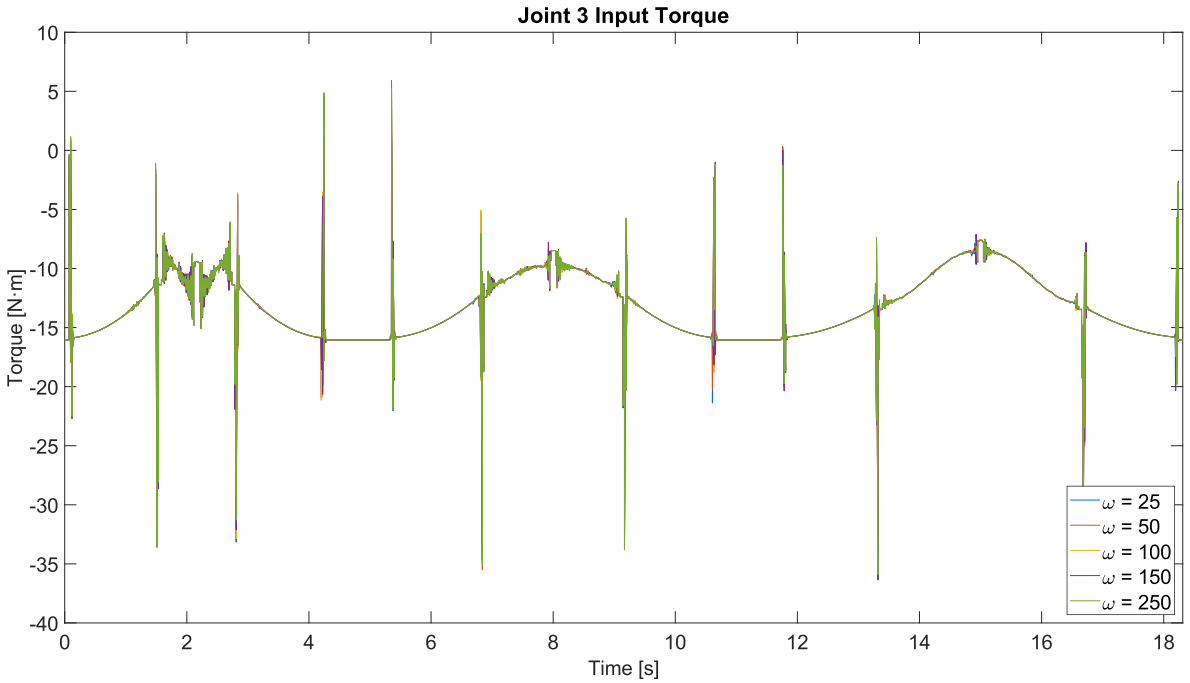

*Plot 3-11. CTC Controller with RobotStudio trajectory. Close-up on input torque in joint 1.*



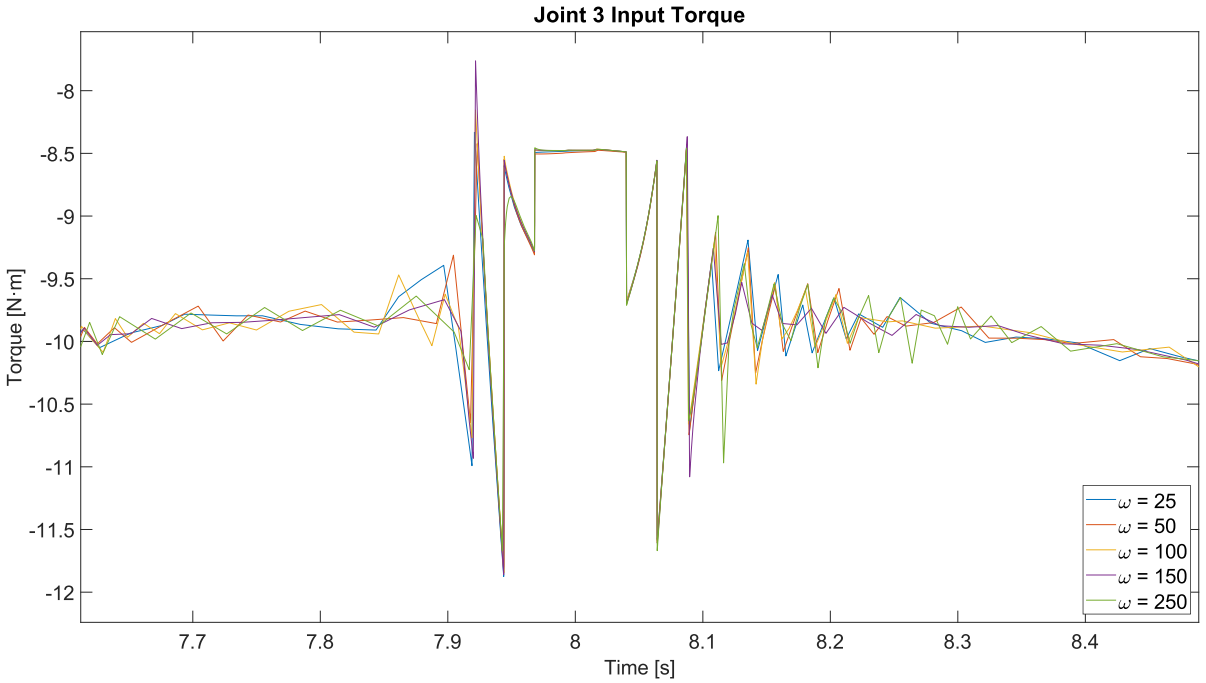

*Plot 3-12. CTC Controller with RobotStudio trajectory. Input torque in joint 2.*



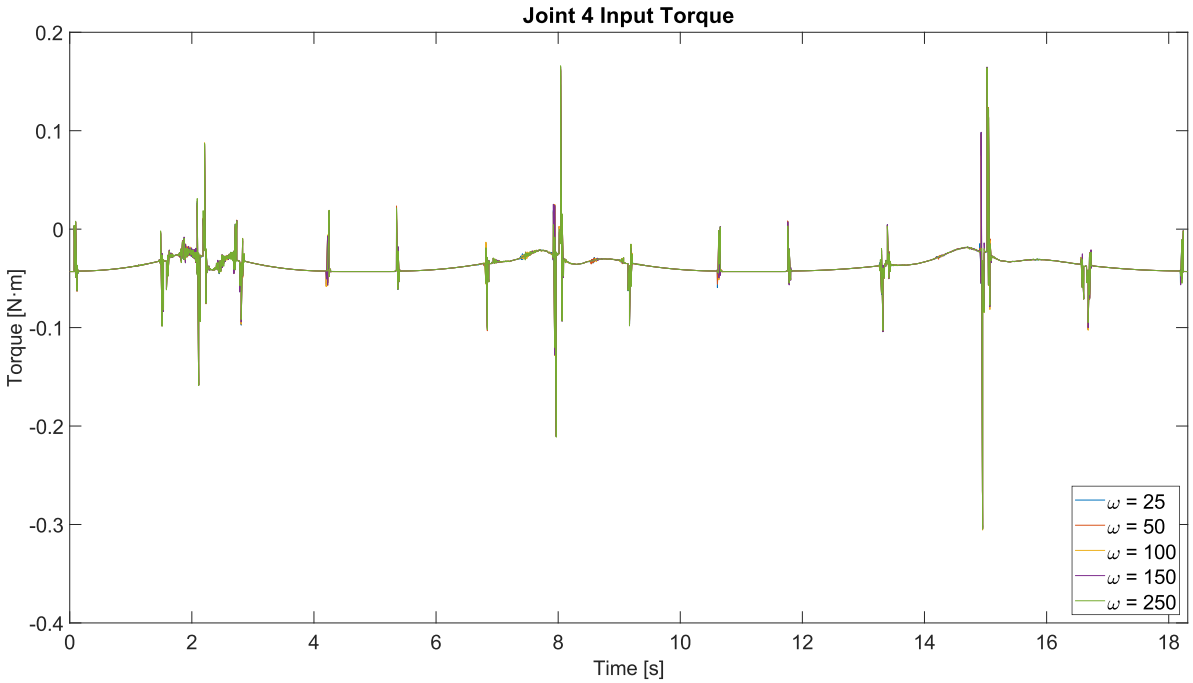

*Plot 3-13. CTC Controller with RobotStudio trajectory. Close-up on input torque in joint 2.*



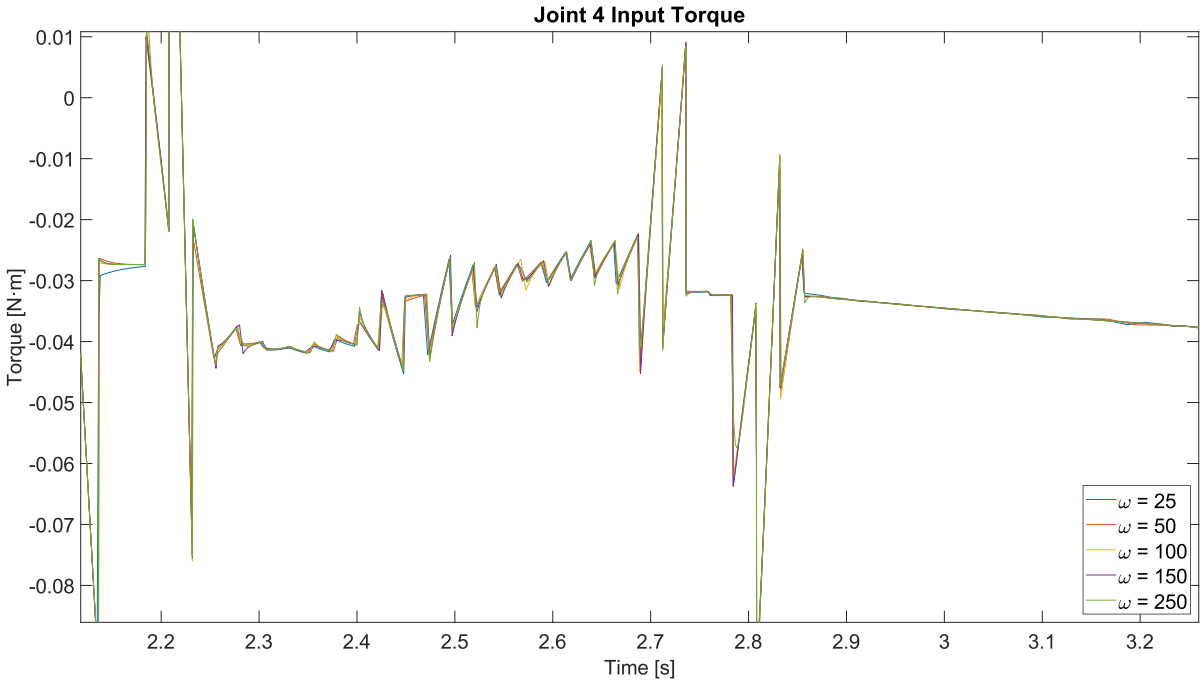

*Plot 3-14. CTC Controller with RobotStudio trajectory. Input torque in joint 3.*



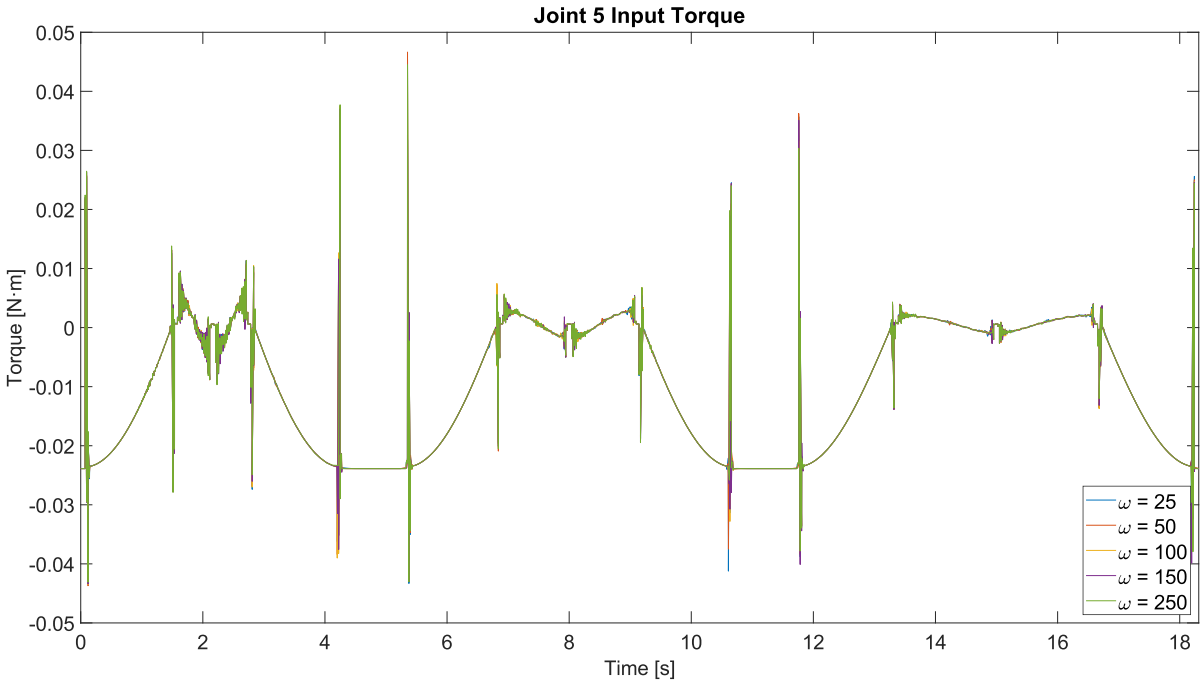

*Plot 3-15. CTC Controller with RobotStudio trajectory. Close-up on input torque in joint 3.*



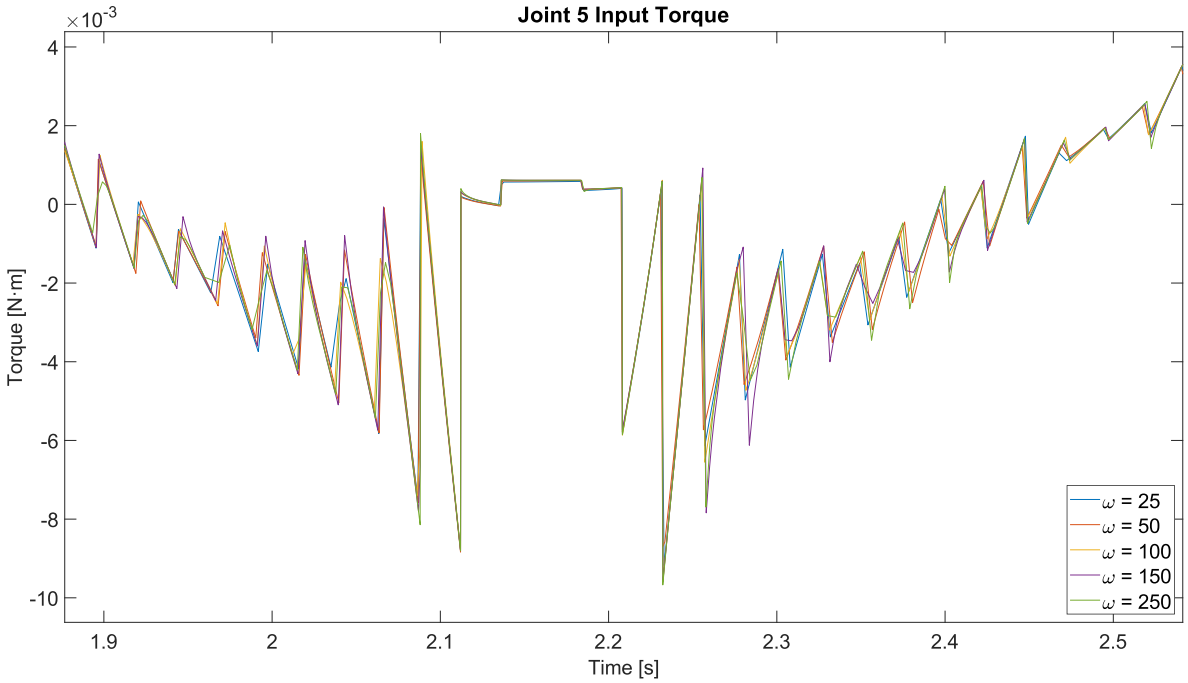

*Plot 3-16. CTC Controller with RobotStudio trajectory. Input torque in joint 4.*



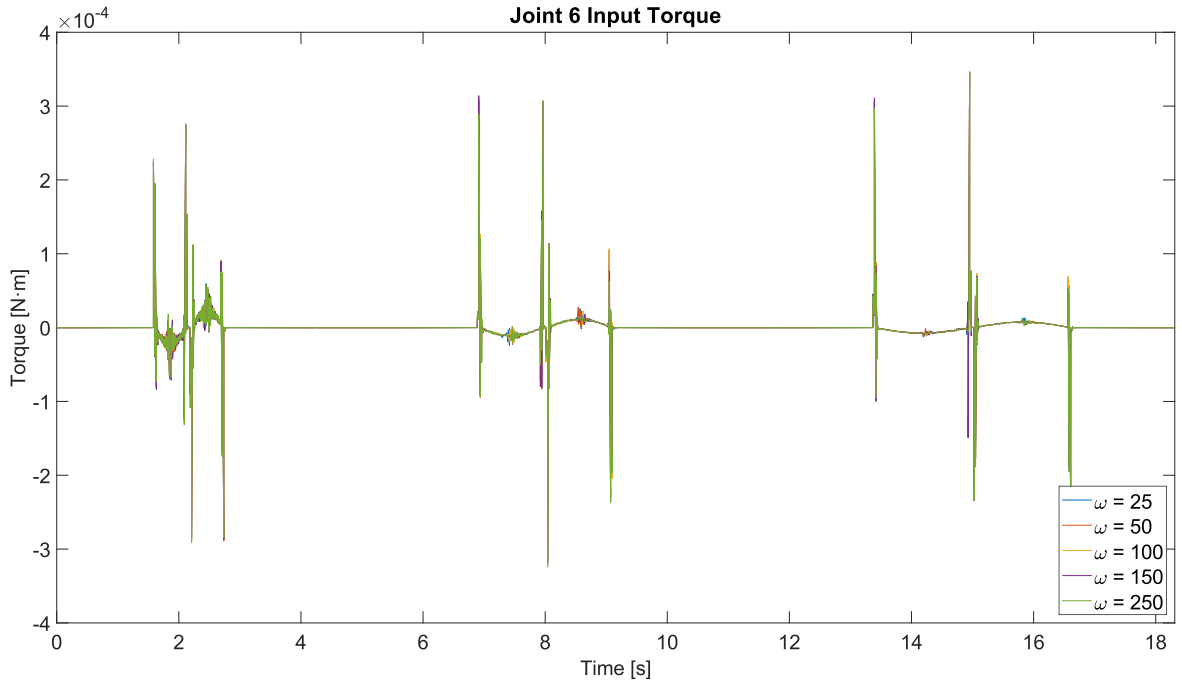

*Plot 3-17. CTC Controller with RobotStudio trajectory. Close-up on input torque in joint 4.*



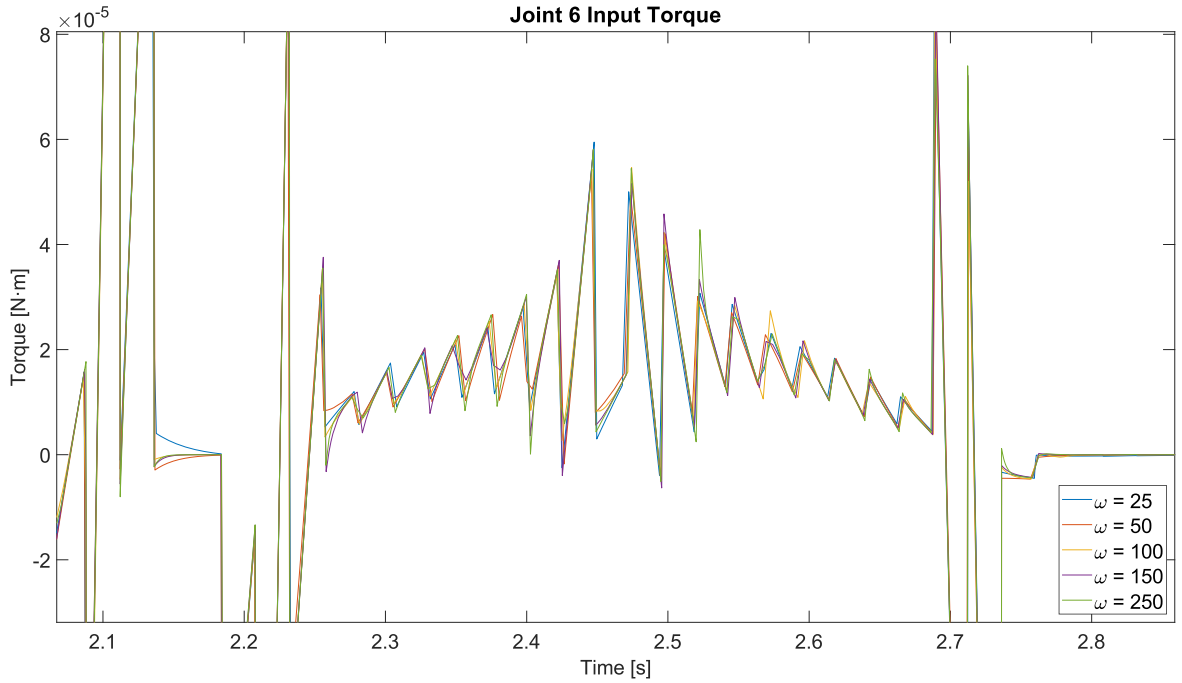

*Plot 3-18. CTC Controller with RobotStudio trajectory. Input torque in joint 5.*




*Plot 3-19. CTC Controller with RobotStudio trajectory. Close-up on input torque in joint 5.*




*Plot 3-20. CTC Controller with RobotStudio trajectory. Input torque in joint 6.*

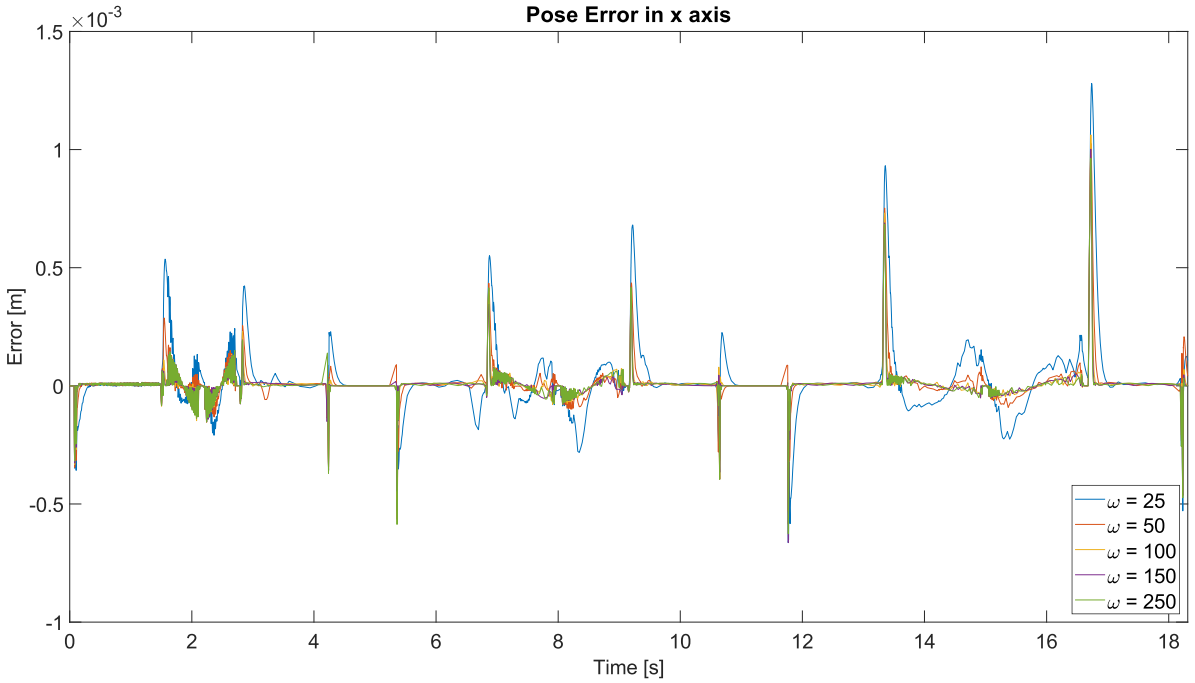



*Plot 3-21. CTC Controller with RobotStudio trajectory. Close-up on input torque in joint 6.*

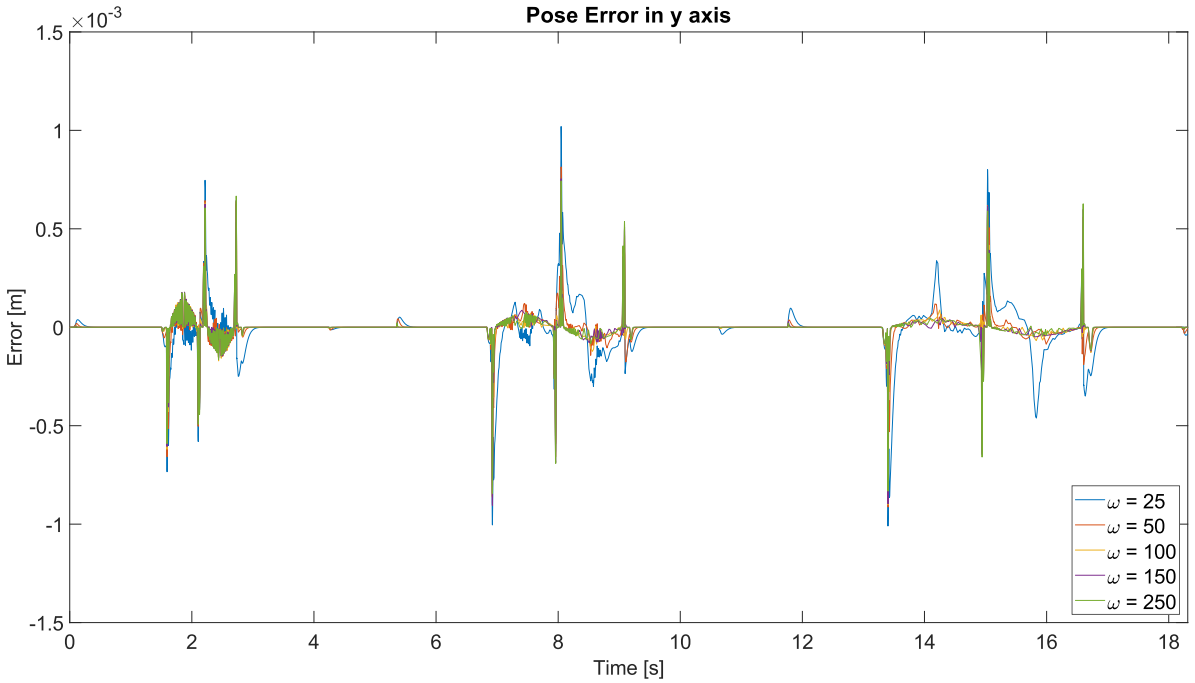
As expected, the higher the bandwidth, the more aggressive the control signal becomes. Nevertheless, there is a shared oscillatory behavior in all bandwidths, most likely caused by the originally oscillatory behavior of the input pose acceleration signal.

### *3.5.1.2 Pose errors*

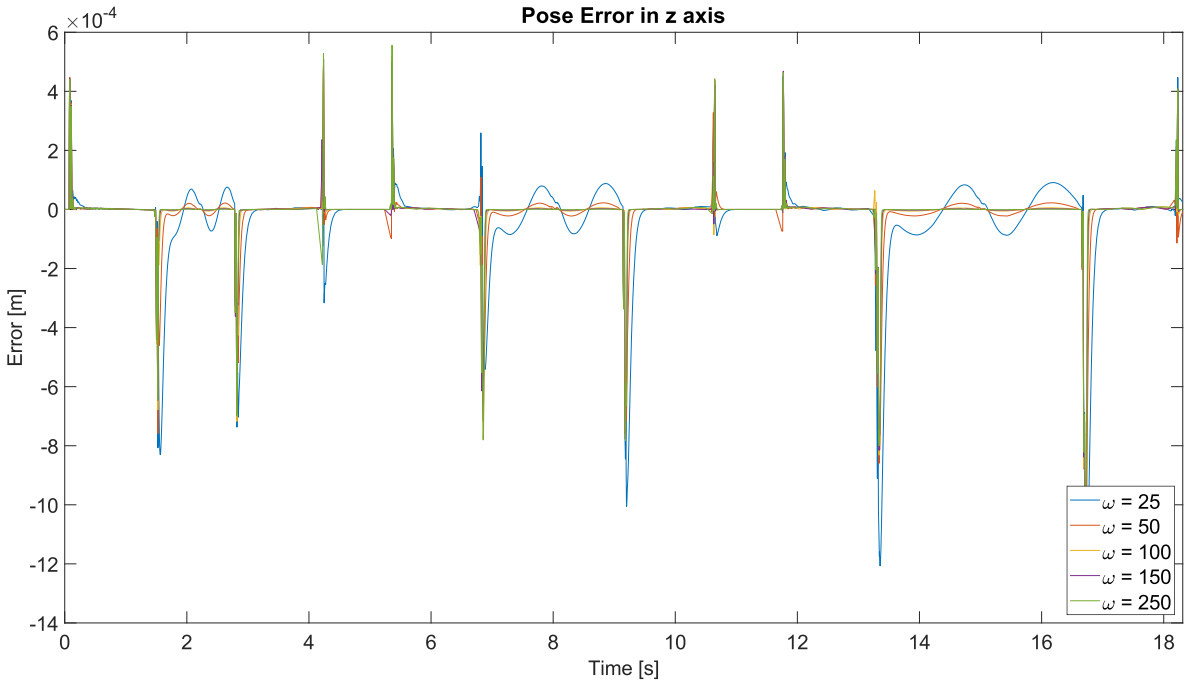
Each of the pose component errors has been represented in an individual plot.



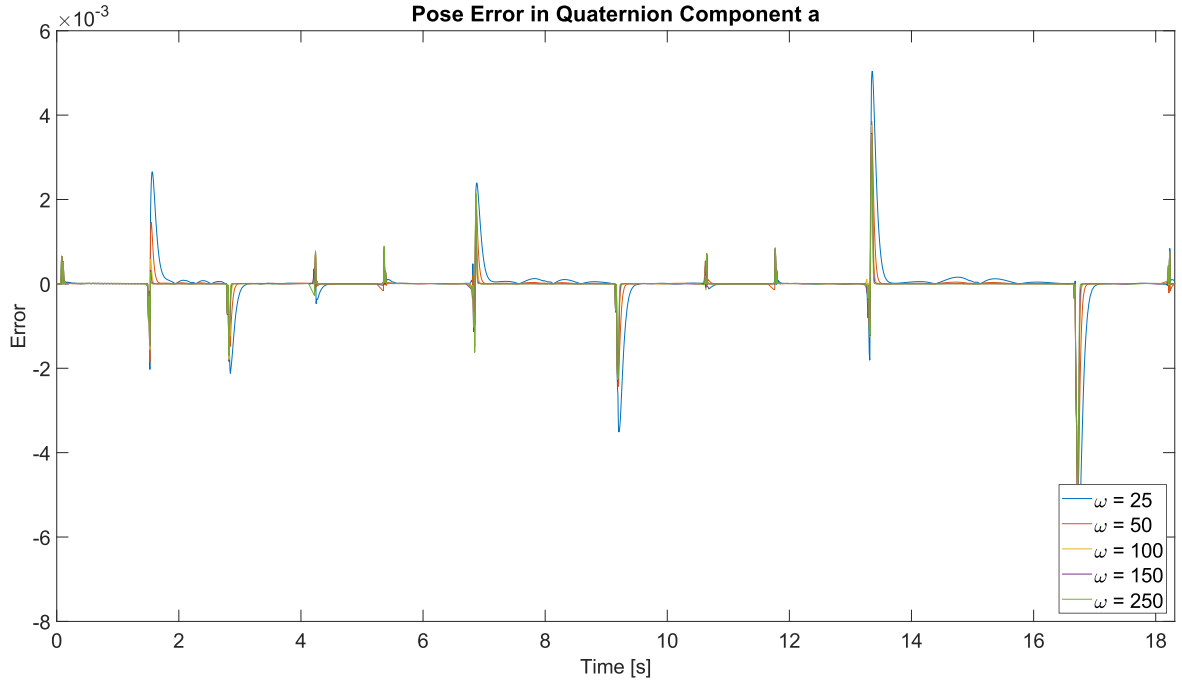

*Plot 3-22. CTC Controller with RobotStudio trajectory. Committed error in $x$-axis.*



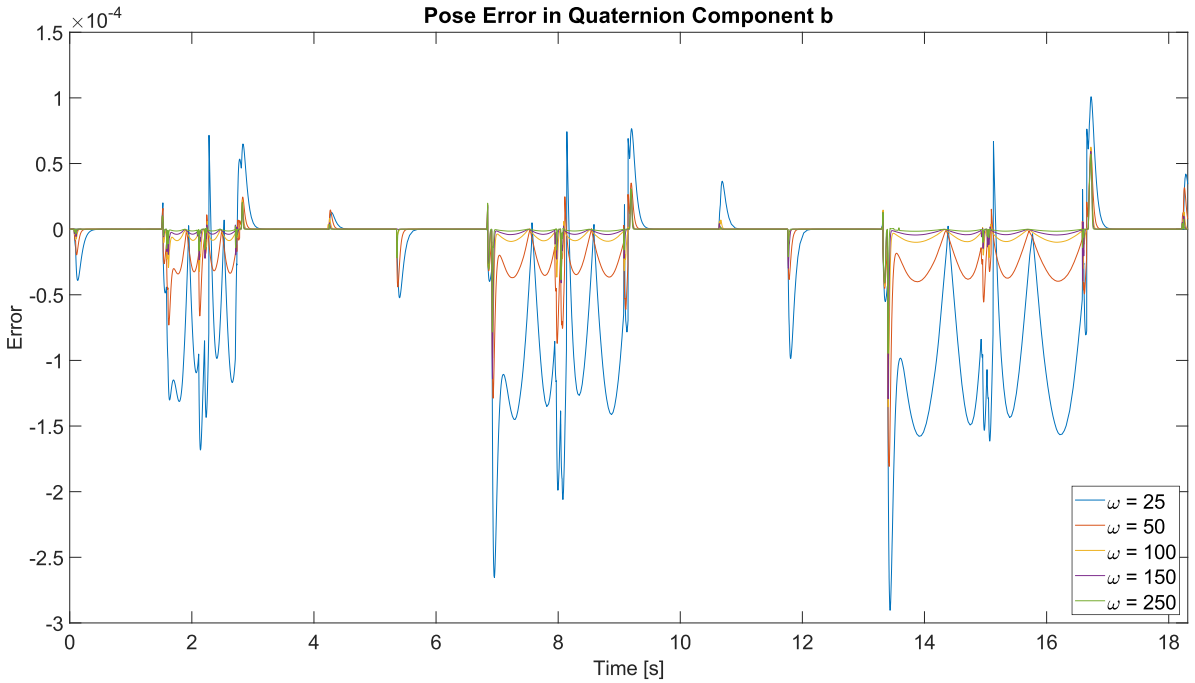

*Plot 3-23. CTC Controller with RobotStudio trajectory. Committed error in $y$-axis.*



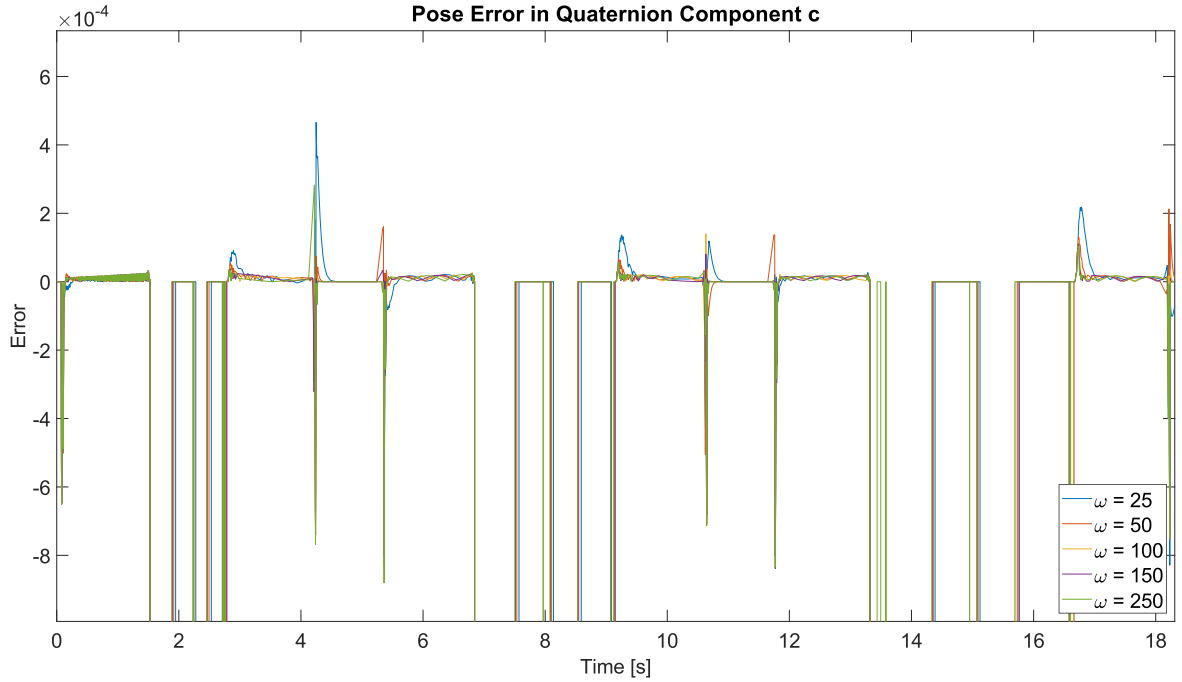

*Plot 3-24. CTC Controller with RobotStudio trajectory. Committed error in z-axis.*



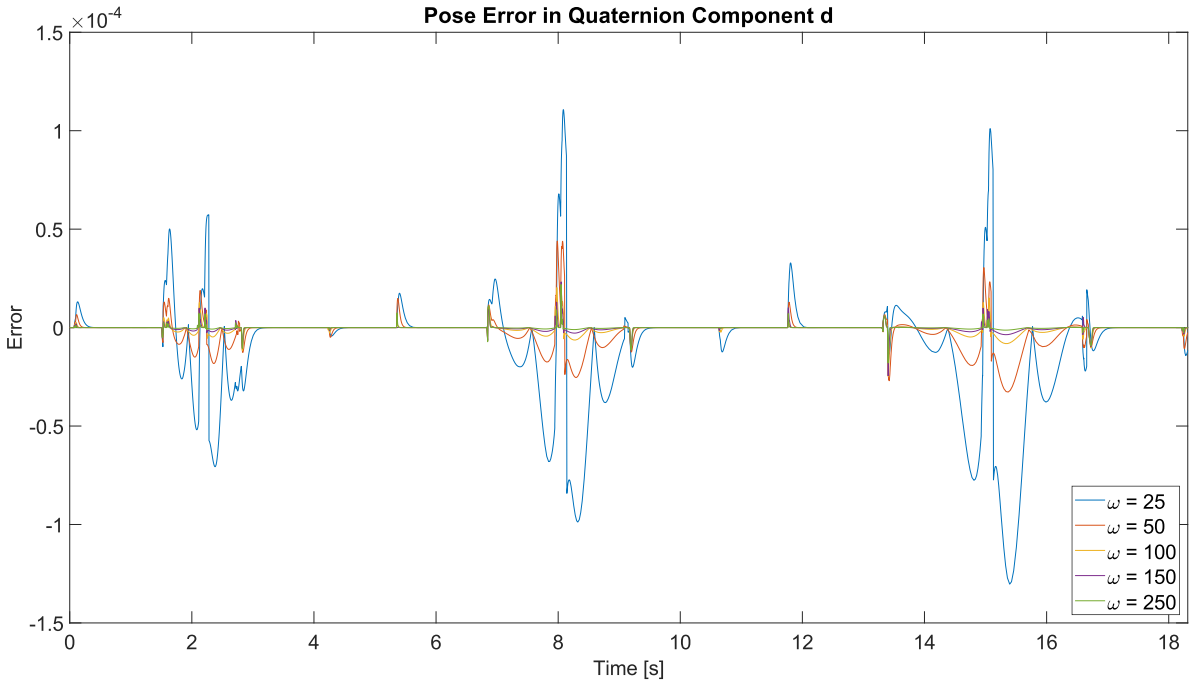

*Plot 3-25. CTC Controller with RobotStudio trajectory. Committed error in quaternion component a.*




*Plot 3-26. CTC Controller with RobotStudio trajectory. Committed error in quaternion component* **b**.




*Plot 3-27. CTC Controller with RobotStudio trajectory. Committed error in quaternion component* **c**. *Bursts in signals are caused by the equivalence -1=1 in quaternion components.*




*Plot 3-28. CTC Controller with RobotStudio trajectory. Committed error in quaternion component* $d$.

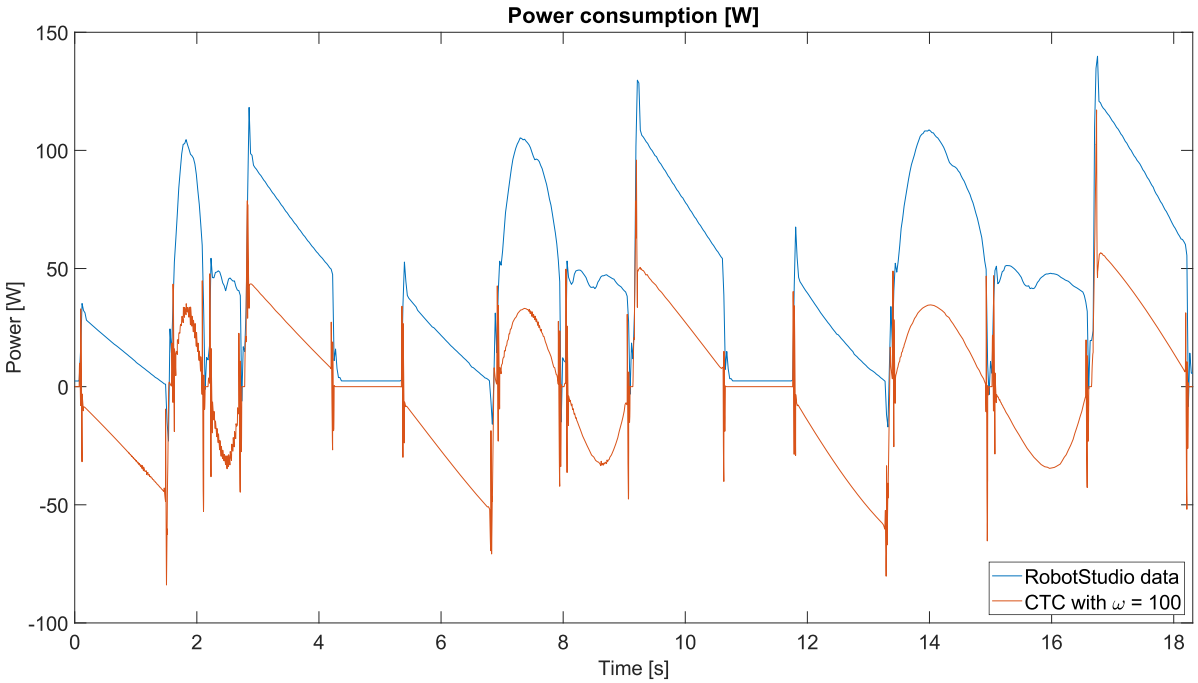
The spatial errors committed range from zero to 1 mm for all tested bandwidths. Most of the time much lower. Quaternion components also count with very low errors along the simulation with maximum errors happening at quaternion component $a$.

#### *3.5.1.3 Power consumption*

RobotStudio also offers the total power consumption of the robot. Although the individual consumption of each torque is unknown, it still helps to get an impression of the likeness of the obtained dynamic model and the RobotStudio dynamic model. The power function has the following form:

$$P(t) = \tau(t) \cdot \omega(t) \tag{3.7}$$

Setting $\omega$ to 100:




*Plot 3-29. Comparison between the power consumptions of the dynamic model and the RobotStudio model.*

There is some resemblance between both plots, however there are two main differences: some acute offset appears during the whole test and the second half of the circular movement seems to produce positive power instead of negative (dextrorotatory instead of levorotatory direction of rotation).

The power needed by the RobotStudio model results higher, which makes sense given that many phenomena like friction, backlash, mechanical noise among others were not included in the dynamic model calculated in this master's thesis. After the addition of these to the dynamic model, the differences between both power consumption should eventually reach zero.

It has been proven that a Computed Torque Controller can offer a precise control solution considering inputs coming from a very accurate simulation of the robot arm ABB IRB 140.

However, the quality of the extracted data from RobotStudio leaves a lot to be desired. Its discontinuity and incapacity of offering velocities and accelerations of the end effector leads to wobbly input torque signals that do not seem doable for implementation in a real system.

Therefore, a different approach to the control problem was proposed: drafting the pose vector signals directly. This vector would be time-continuous so it would not suffer from the same complications as the RobotStudio signals. Its derivatives could be computed straightforwardly and should suppose smoother torque control signals. The step-by-step implementation is presented in the next section.

### 3.5.2 Full system model with a Trajectory Generator block as input and a Computed Torque Controller

The proposed structure for the Trajectory Generator block is as follows (Figure 3-54):

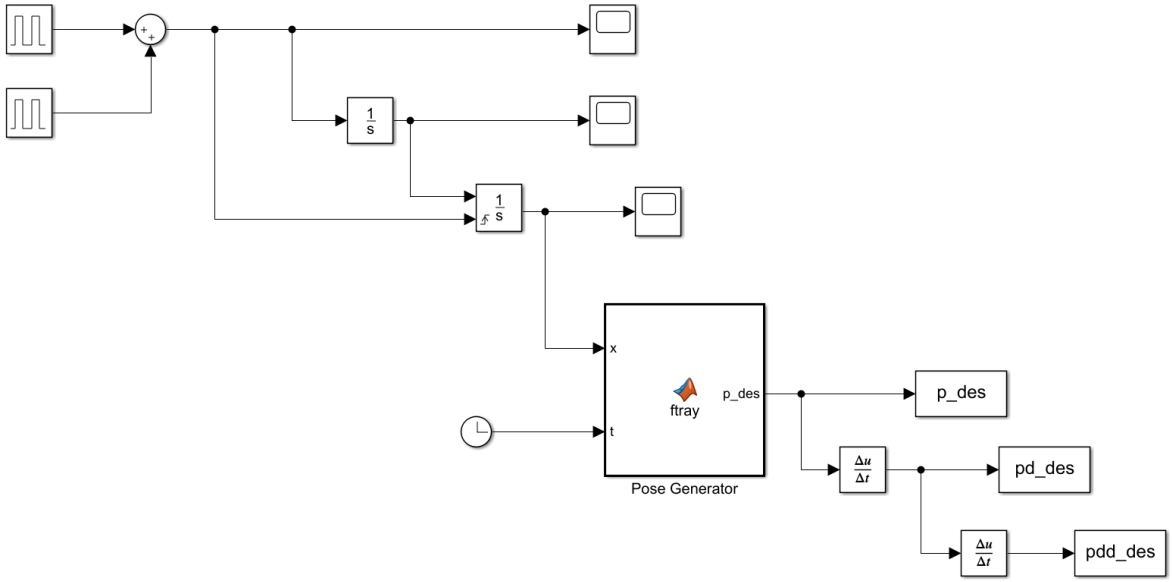
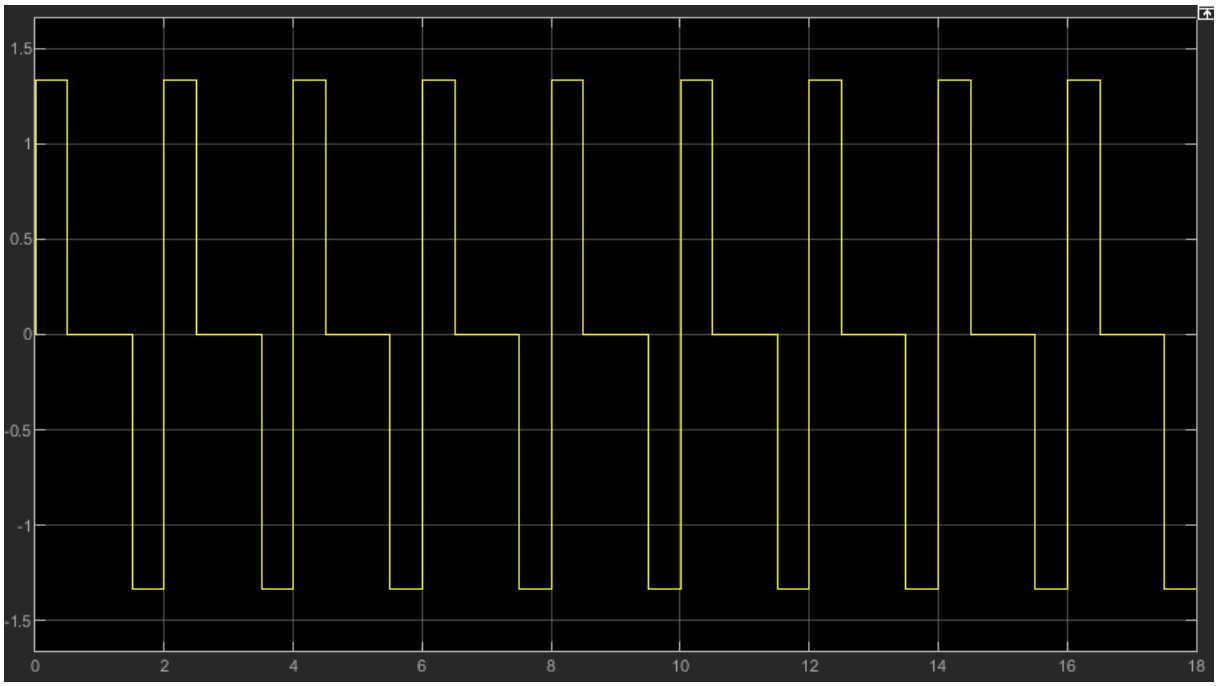



*Figure 3-54. Trajectory Generator schematic.*

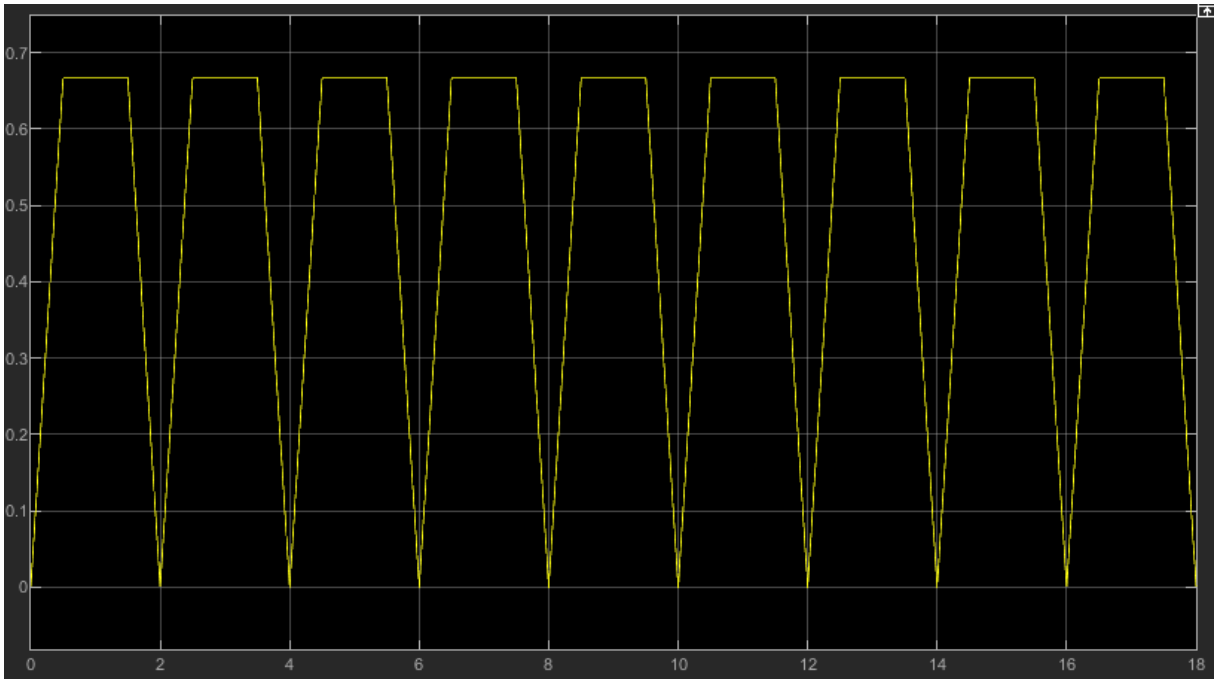
Altogether, the block draws 3 different signals, the desired pose vector, the desired pose velocities and the desired pose accelerations. The trajectories themselves will be the same as in the previous section (Table 3-8), although with different execution time: 2 seconds per move.

It starts generating periodic acceleration profiles through two Pulse Generator blocks. These profiles are integrated twice, obtaining in this manner the velocity and displacement profiles. The displacement integrator block must be reset to avoid signal overlapping after the first acceleration profile. The overall goal is to achieve a displacement profile that reaches the unit at the end of the execution time and then adapt it to fit the geometrical requirements of every move.

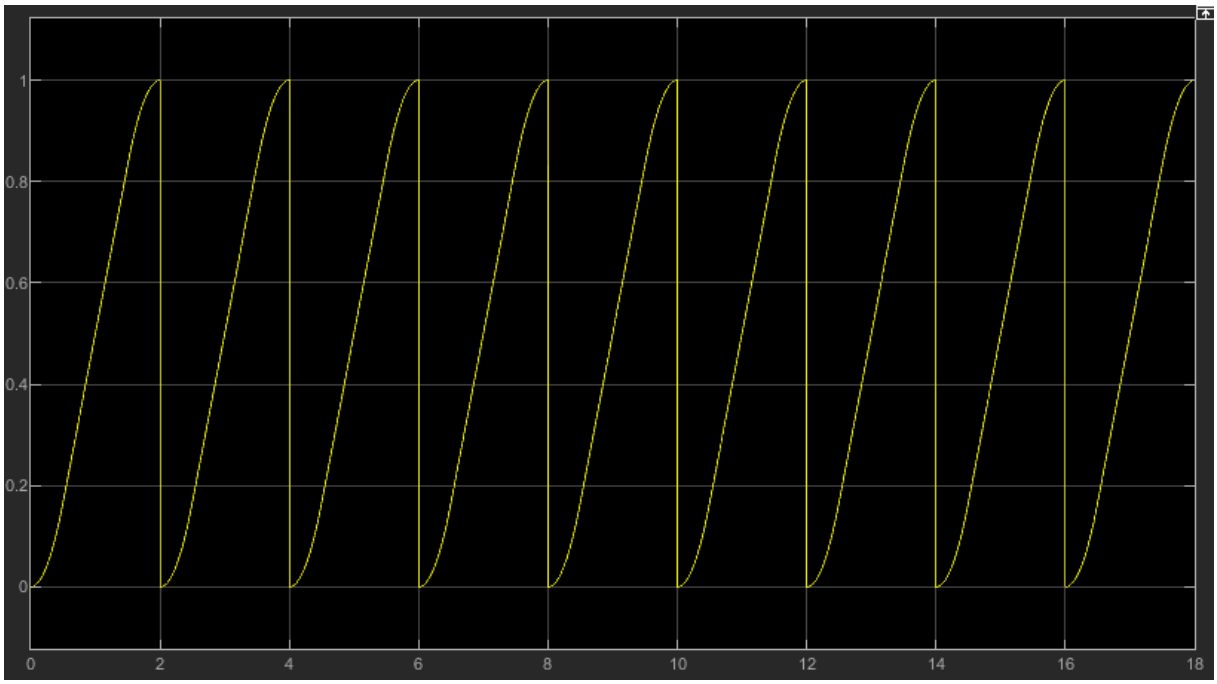
Next, all three profiles are introduced into a MATLAB Function block, which contains all pose data and creates a continuous and proportional trajectory, forming a vector of poses. Deriving these, the final output results would be a vector of poses, a vector of pose velocities and a vector pose accelerations that will serve as input for the robot system.




*Plot 3-30. Acceleration profiles of the Trajectory Generator.*

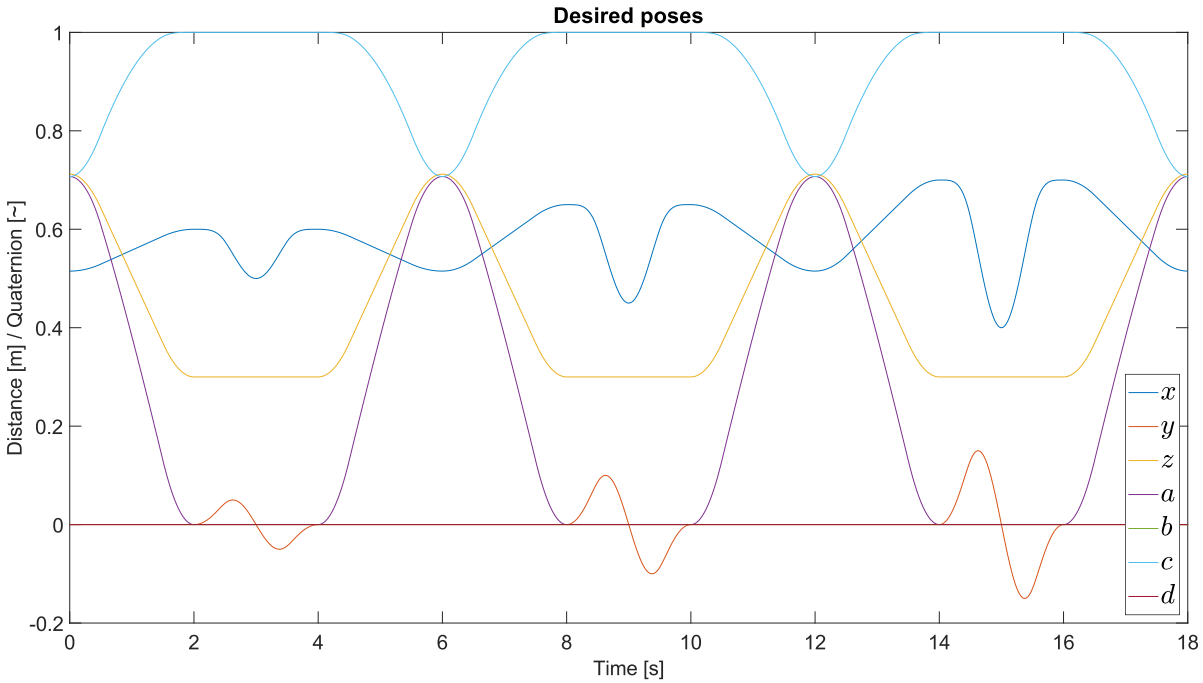



*Plot 3-31. Velocity profiles of the Trajectory Generator.*

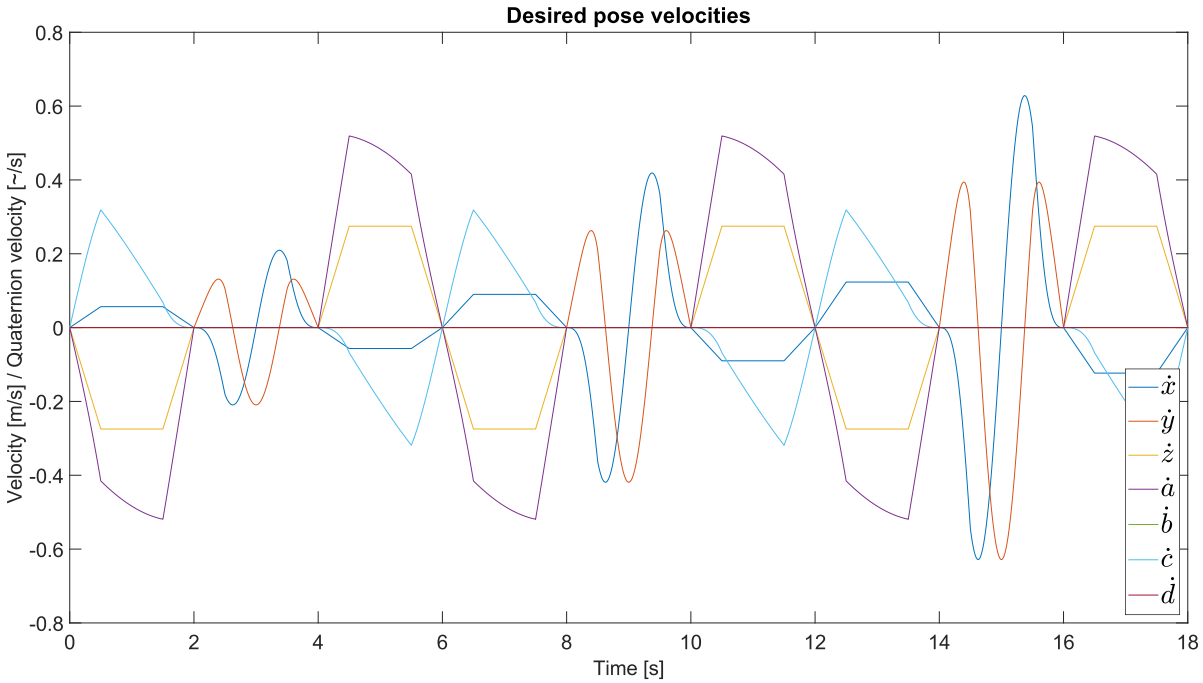



*Plot 3-32. Displacement profiles of the Trajectory Generator.*

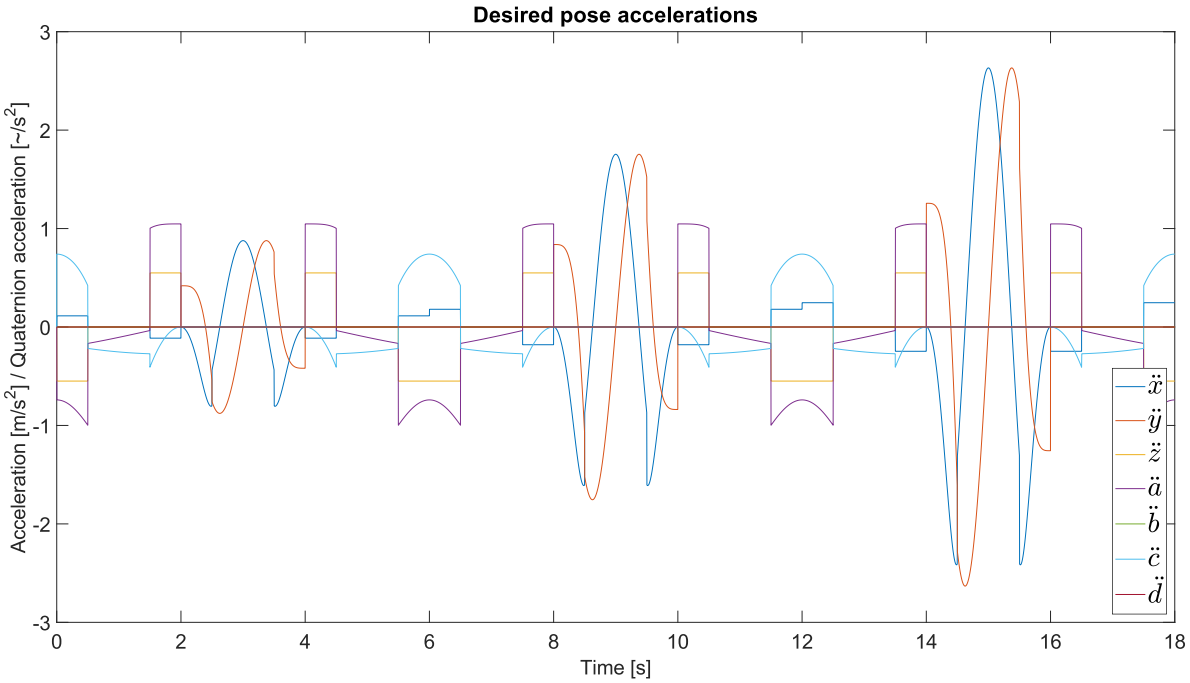
The resulting poses and their derivatives are:




*Plot 3-33. Trajectory Generator poses.*

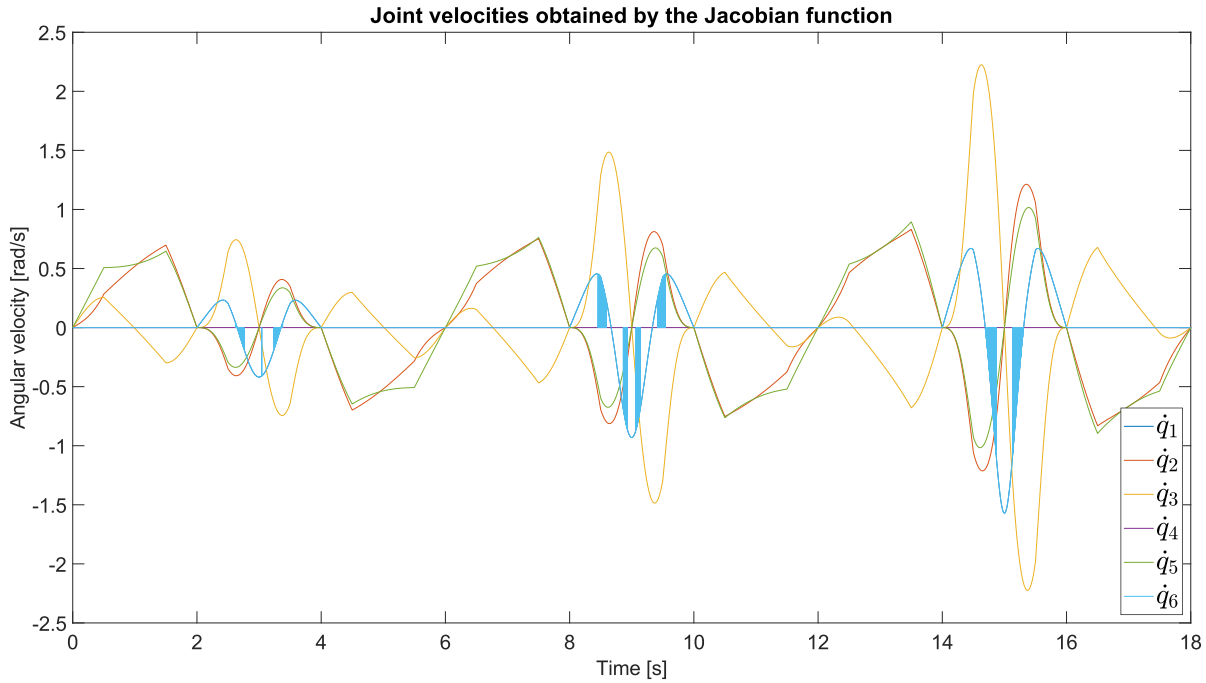



*Plot 3-34. Trajectory Generator pose velocities.*




*Plot 3-35. Trajectory Generator pose accelerations.*

However, after implementing this block in the main schematic, a new issue emerges. Sudden unexpected changes in the Jacobian matrix generate the following joint velocities:



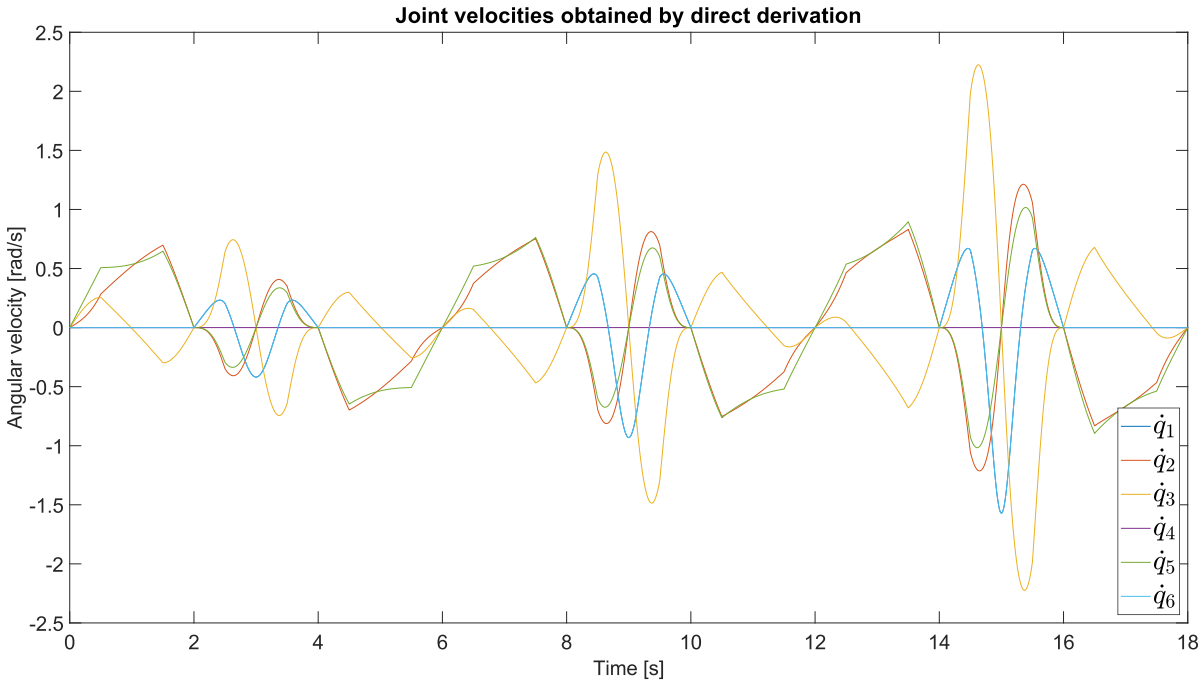

*Plot 3-36. Joint velocities calculated by the Jacobian function.*

In some time intervals, the desired joint velocity for joint number 6 oscillates anomalously. Checking the internal values of the Jacobian function one can observe that after a smooth change of 0.001 seconds (column 2 of Table 3-10) in the values of the joint angles (columns 3 to 8 of Table 3-10), the 4th row of the Jacobian matrix (columns 9 to 14 of Table 3-10) changes the sign of its values.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2167 | 2.1660 | 0.0096 | 0.7468 | -0.0247 | 3.2668e-17 | 0.8486 | 0.0096 | -6.4954e-18 | -0.5000 | -0.5000 | -0.0036 | -0.5000 | -6.5448e-18 |
| 2168 | 2.1670 | 0.0097 | 0.7468 | -0.0246 | 3.4685e-17 | 0.8486 | 0.0097 | 6.5063e-18 | 0.5000 | 0.5000 | 0.0036 | 0.5000 | 6.5063e-18 |

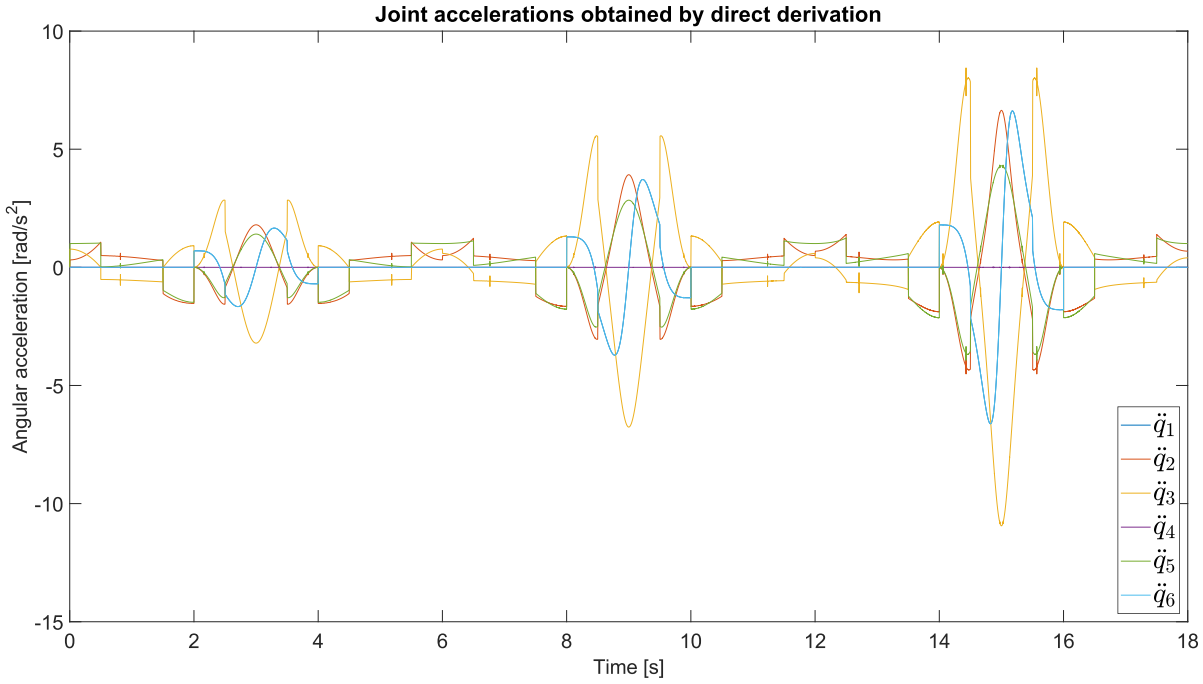
*Table 3-10. Example of undesired behavior of the inverse kinematics function for poses generated by the trajectory generator.*

To solve this situation the joint velocities and joint accelerations for this case have been calculated throughout derivative blocks, as it was done with RobotStudio data.




*Plot 3-37. Joint velocities calculated by straight pre-derivation of the joint angles.*

The result is the removal of the problematic oscillation and generation of a clean vector of joint velocities. Deriving these one more time draws the vector of joint accelerations.




*Plot 3-38. Joint accelerations calculated by straight double pre-derivation of the joint angles.*

These have been stored in .mat files and then called from the main schematic, which now seems as shown in Figure 3-55 (Same as Figure 3-53):

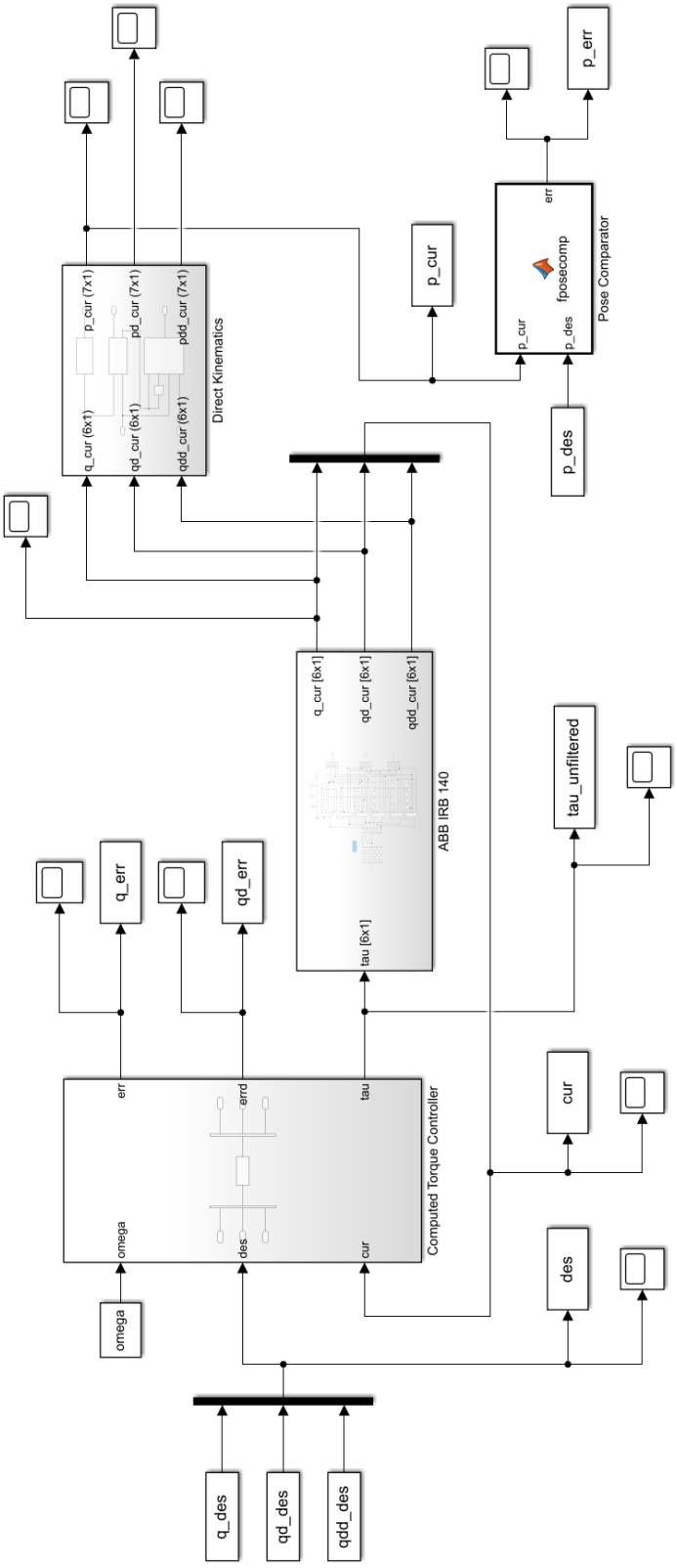
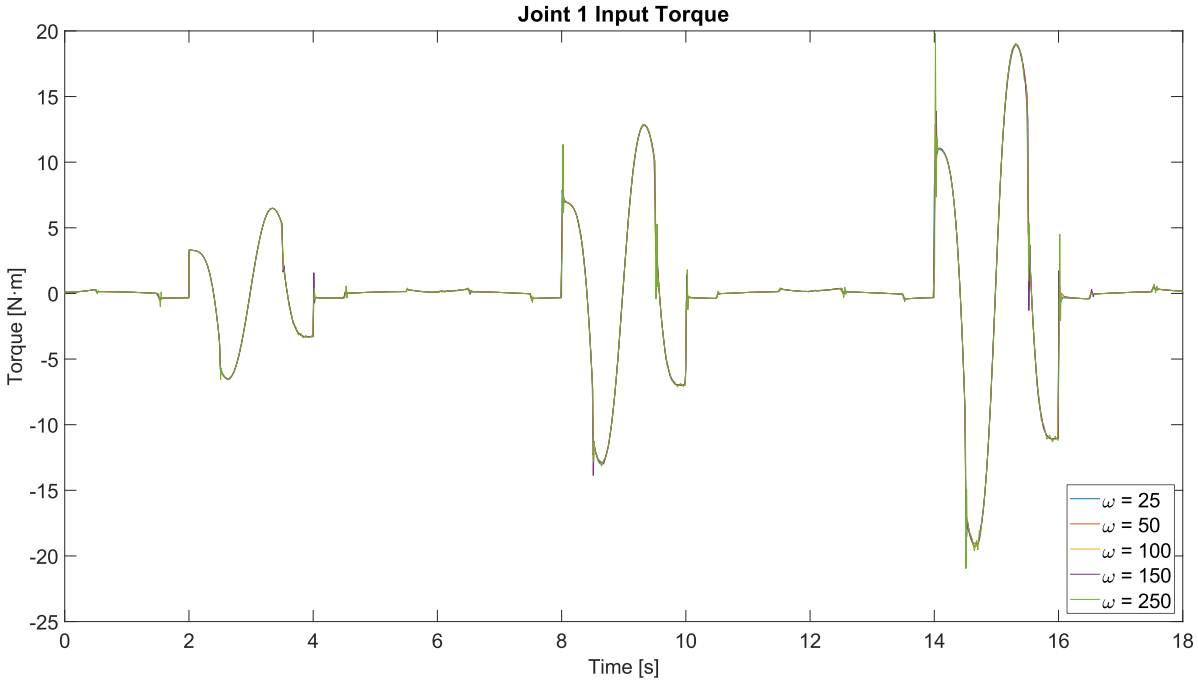



*Figure 3-55. Reduced system schematic with joint coordinates precalculated.*

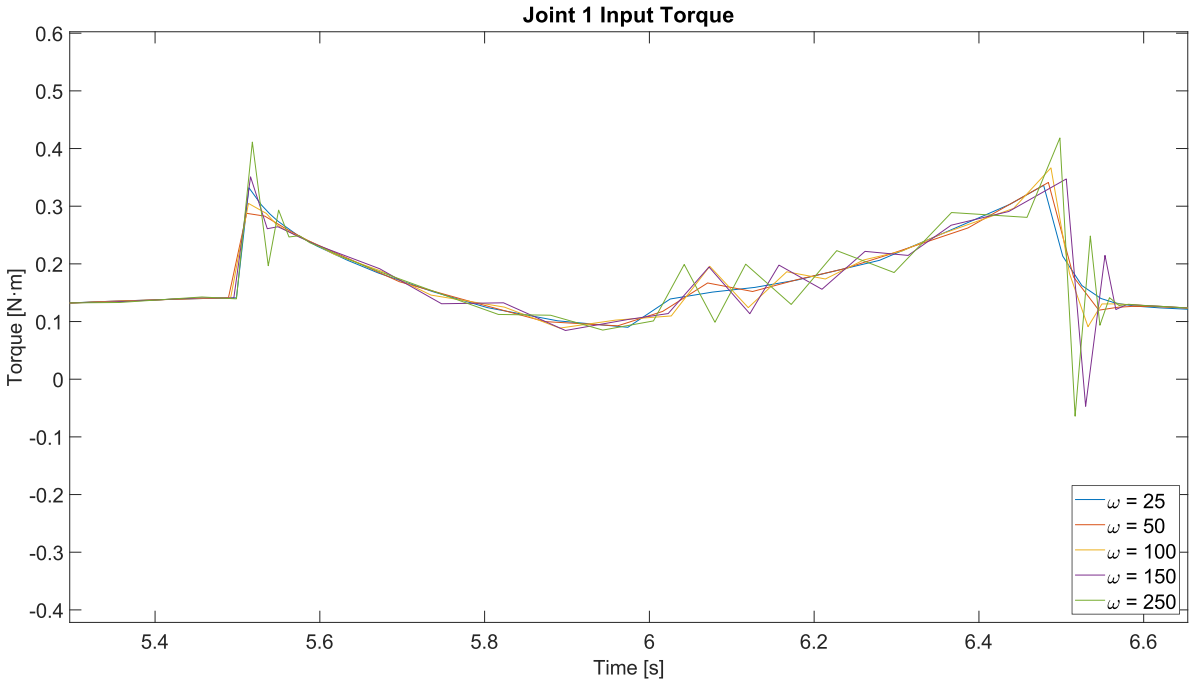
Simulating this schematic, and as the previous section, with many different bandwidths $\omega$, the resulting torques, pose errors and power have been calculated and plotted.

#### *3.5.2.1 Input torques*

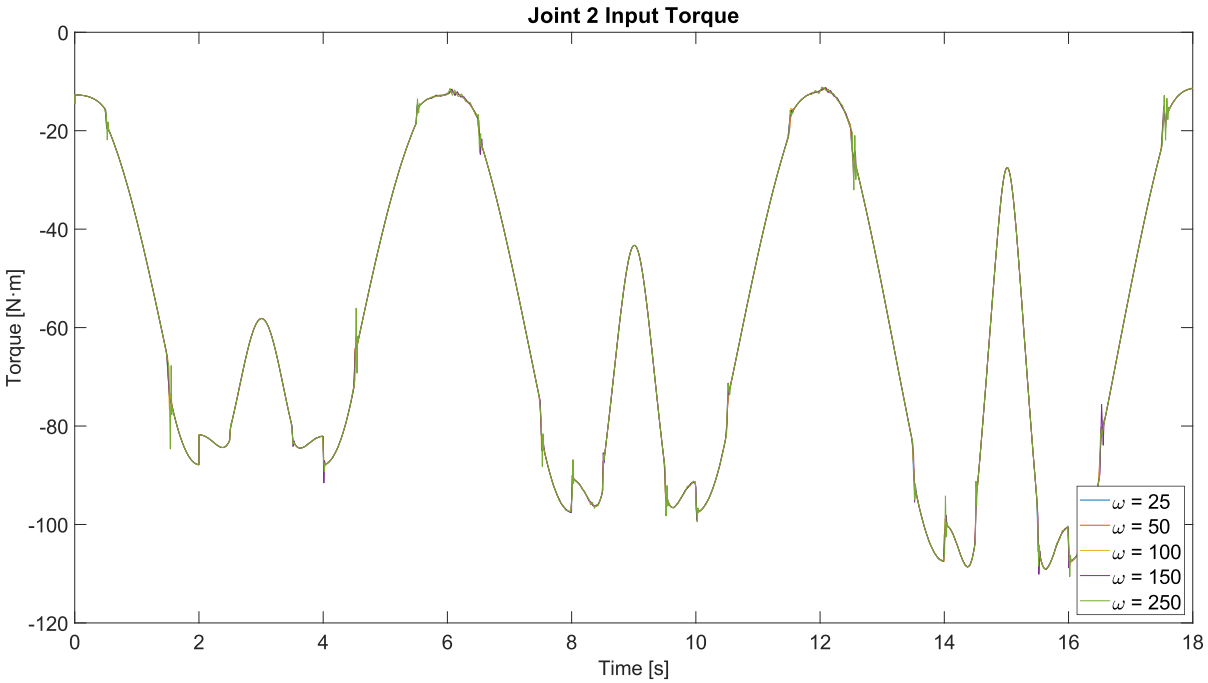



*Plot 3-39. CTC Controller with Trajectory Generator. Input torque in joint 1.*

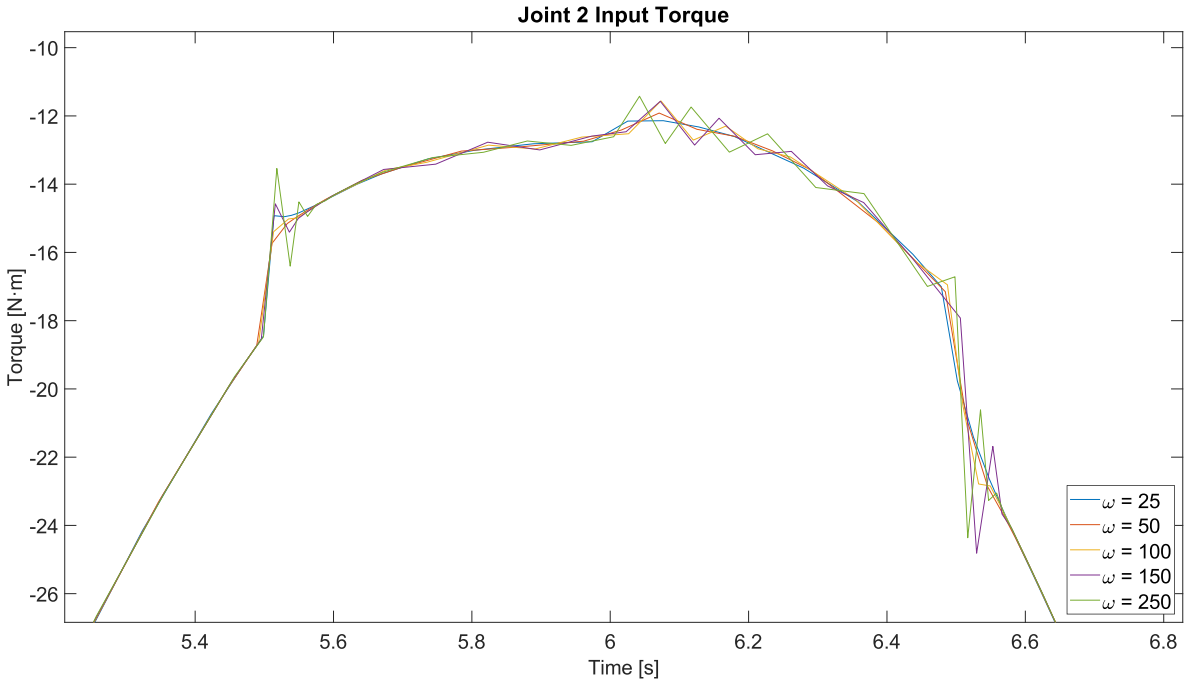
Again, due to the proximity of all signals, a close-up of a section of the plot will be provided.



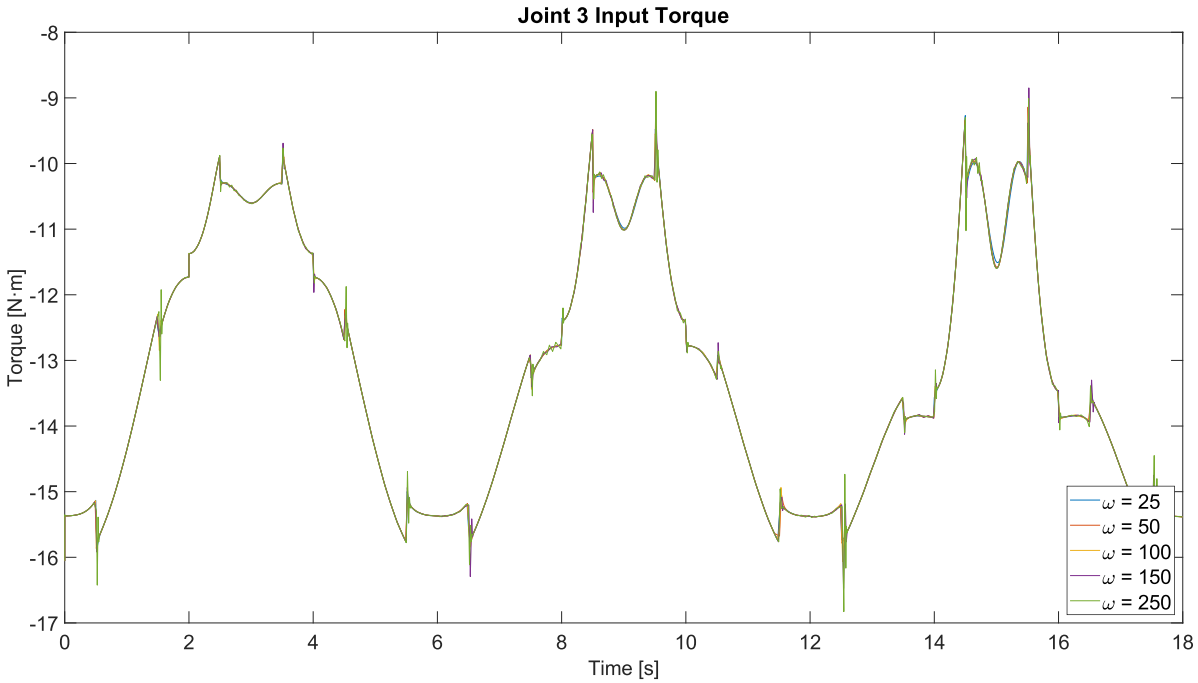

*Plot 3-40. CTC Controller with Trajectory Generator. Close-up on input torque in joint 1.*



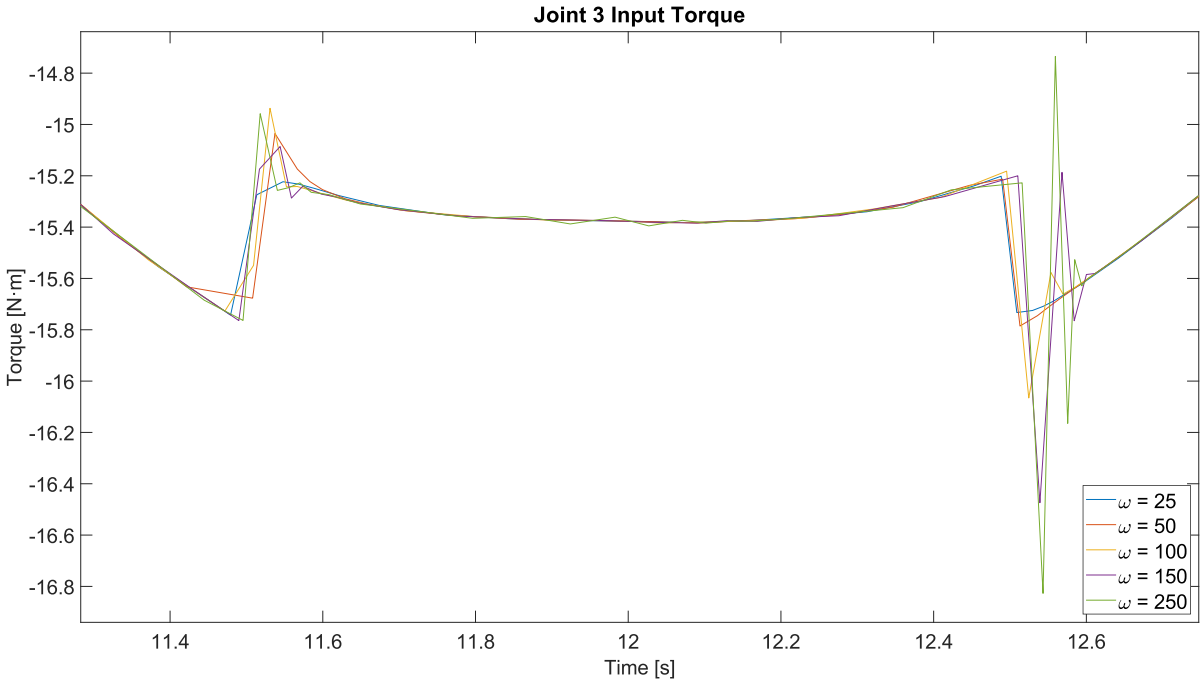

*Plot 3-41. CTC Controller with Trajectory Generator. Input torque in joint 2.*



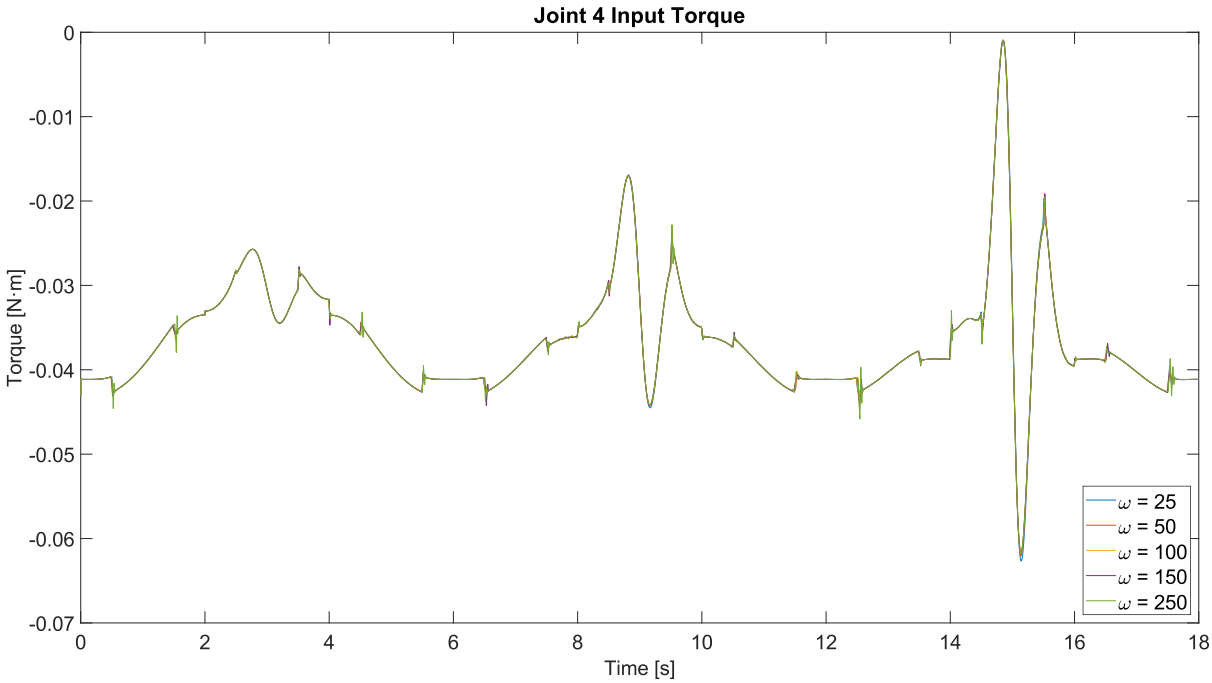

*Plot 3-42. CTC Controller with Trajectory Generator. Close-up on input torque in joint 2.*



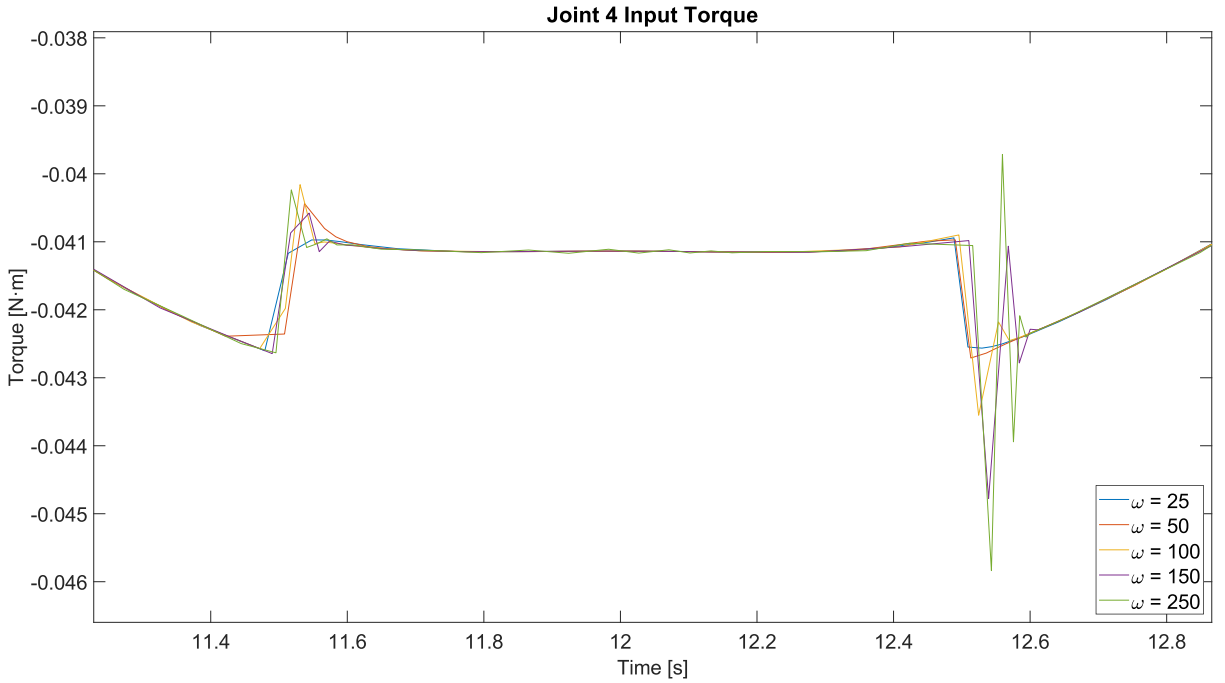

*Plot 3-43. CTC Controller with Trajectory Generator. Input torque in joint 3.*



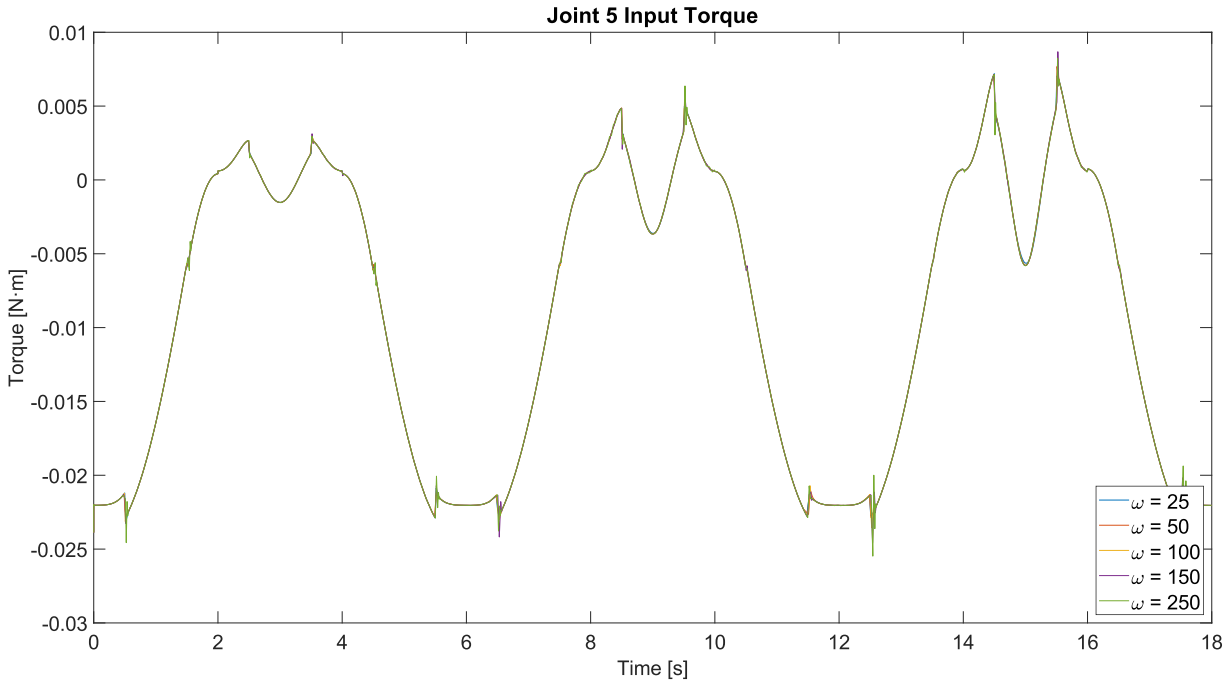

*Plot 3-44. CTC Controller with Trajectory Generator. Close-up on input torque in joint 3.*



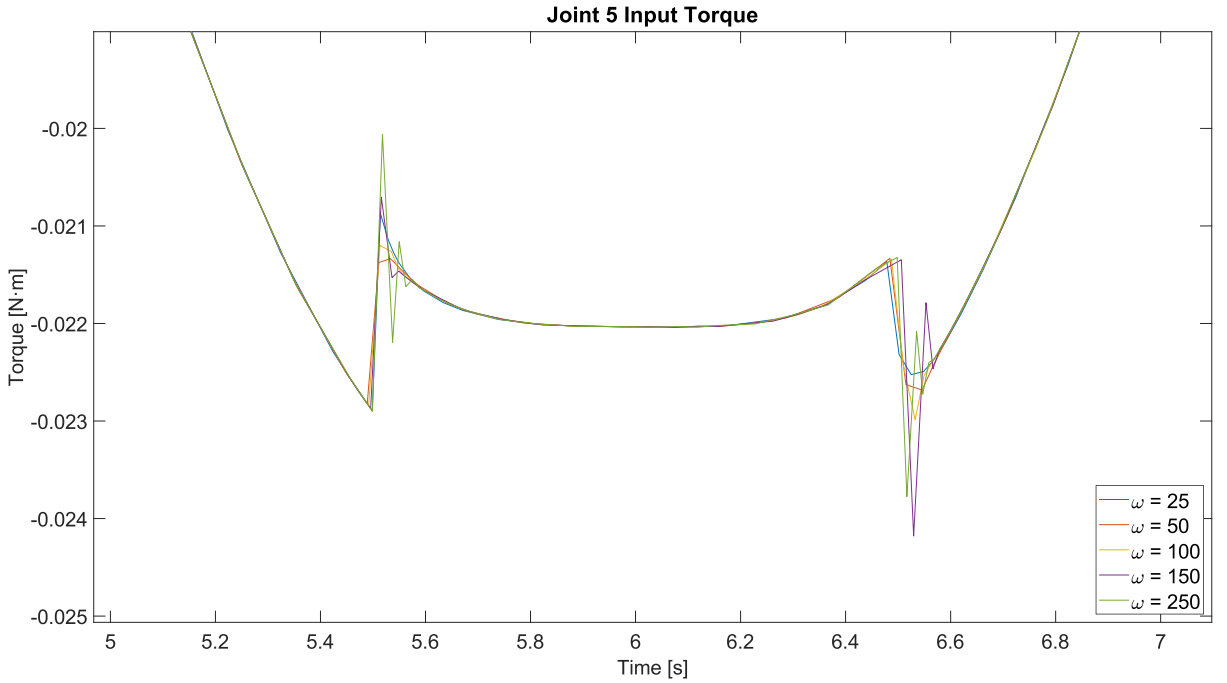

*Plot 3-45. CTC Controller with Trajectory Generator. Input torque in joint 4.*



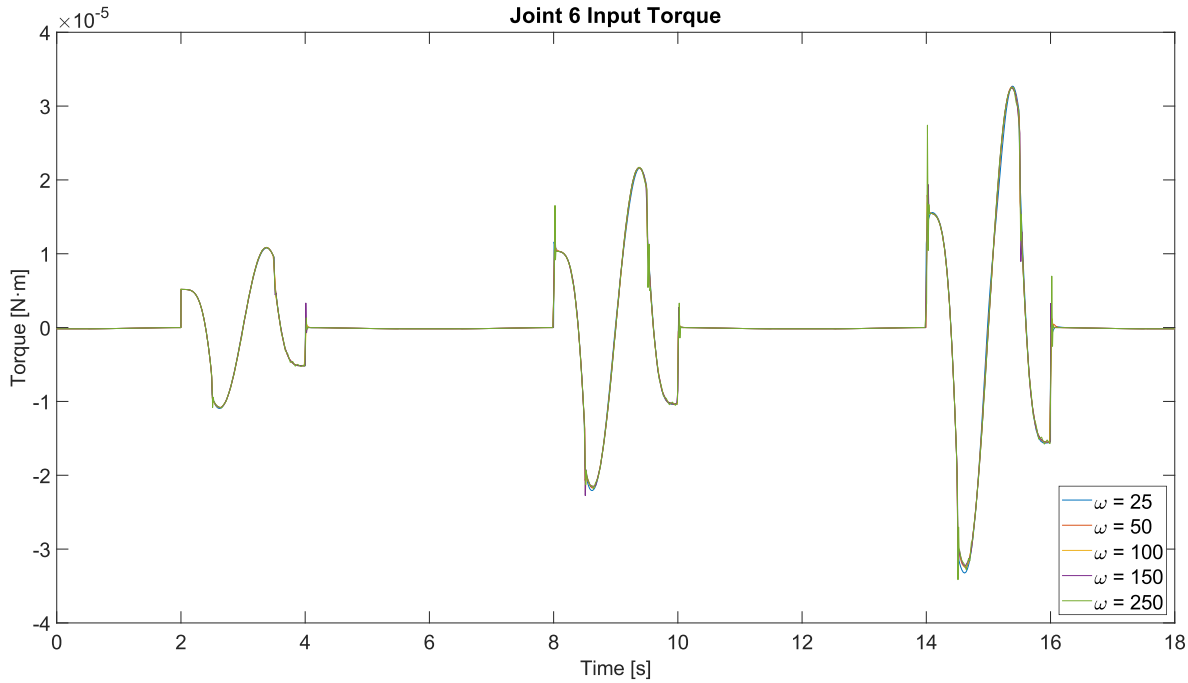

*Plot 3-46. CTC Controller with Trajectory Generator. Close-up on input torque in joint 4.*



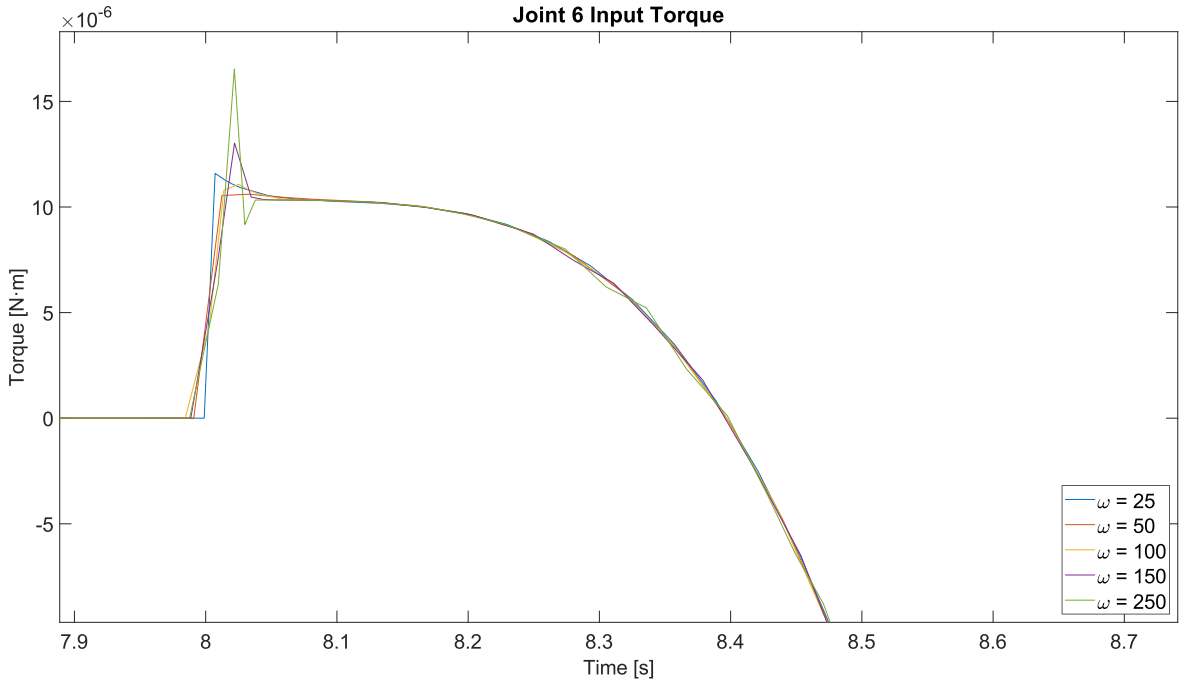

*Plot 3-47. CTC Controller with Trajectory Generator. Input torque in joint 5.*




*Plot 3-48. CTC Controller with Trajectory Generator. Close-up on input torque in joint 5.*



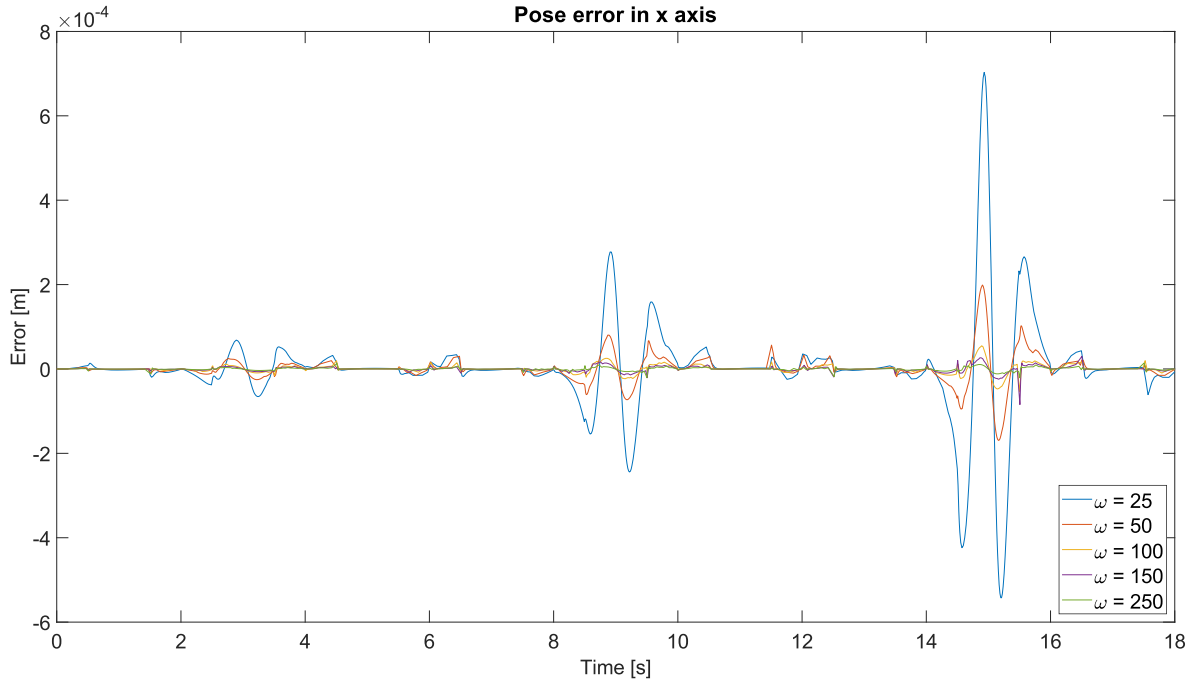

*Plot 3-49. CTC Controller with Trajectory Generator. Input torque in joint 6.*



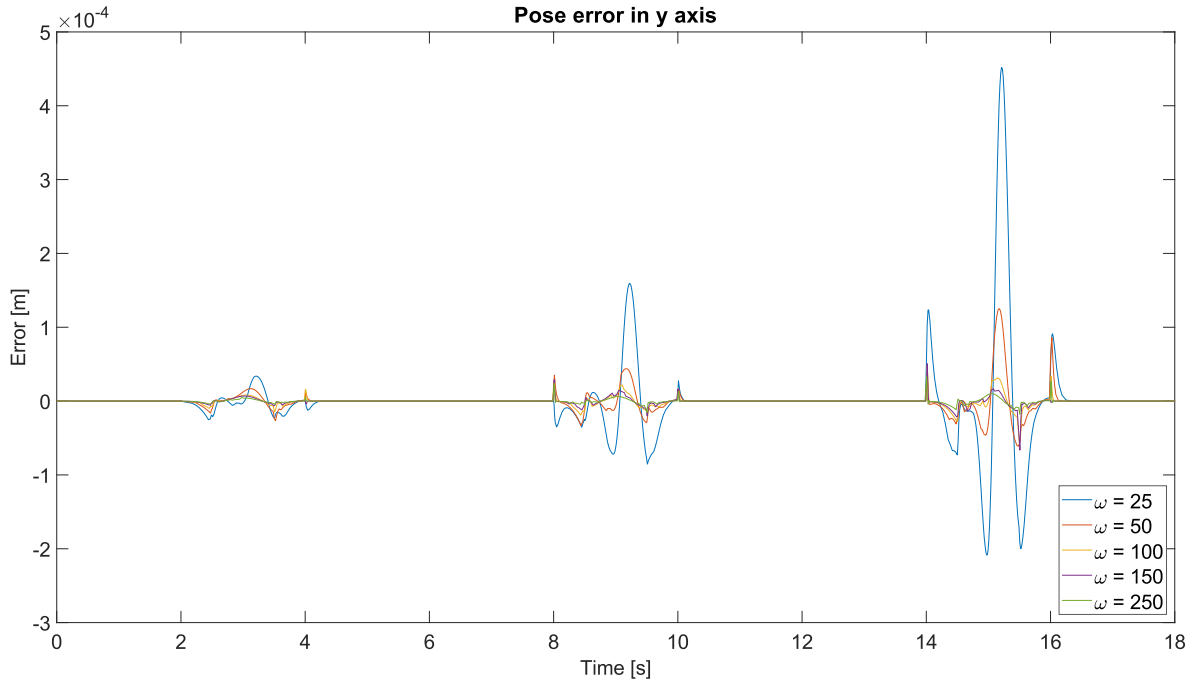

*Plot 3-50. CTC Controller with Trajectory Generator. Close-up on input torque in joint 6.*

#### *3.5.2.2 Pose errors*

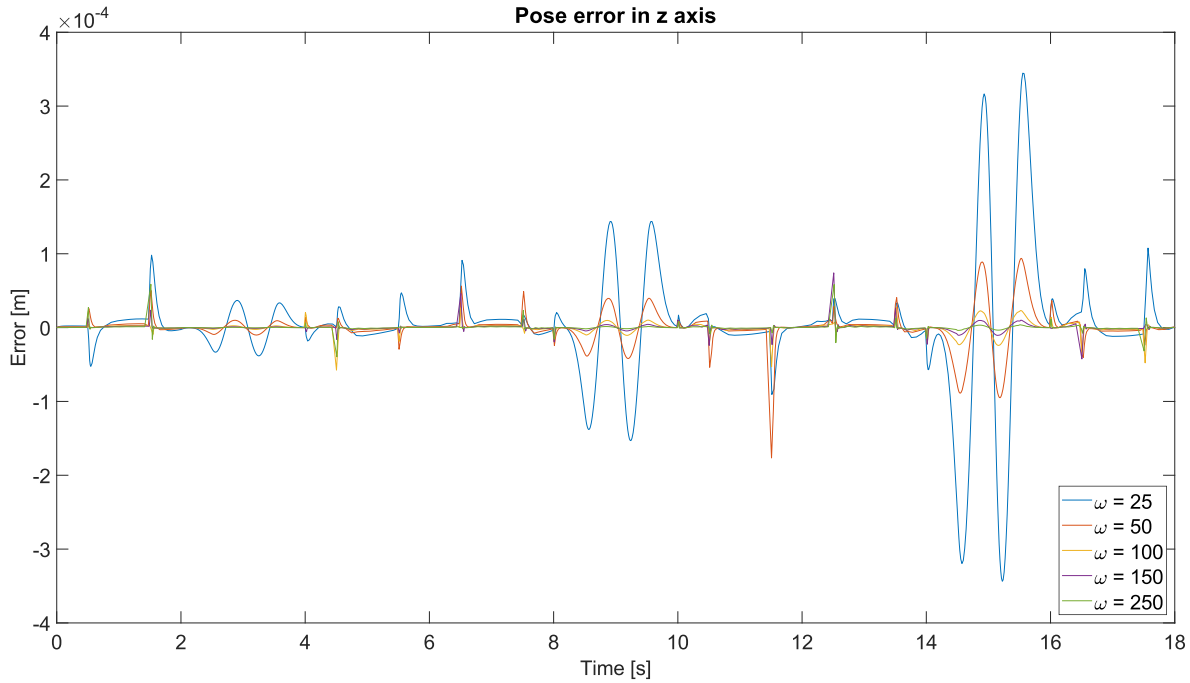
Pose errors in all pose components are:



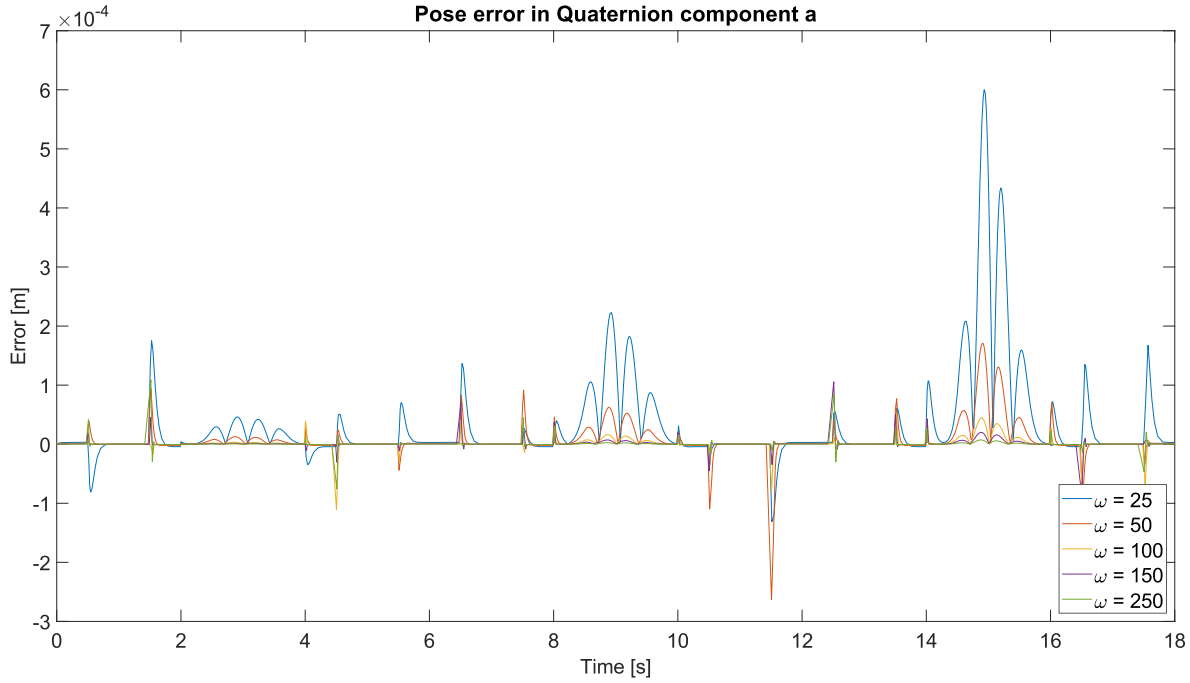

*Plot 3-51. CTC Controller with Trajectory Generator. Committed error in $x$-axis.*



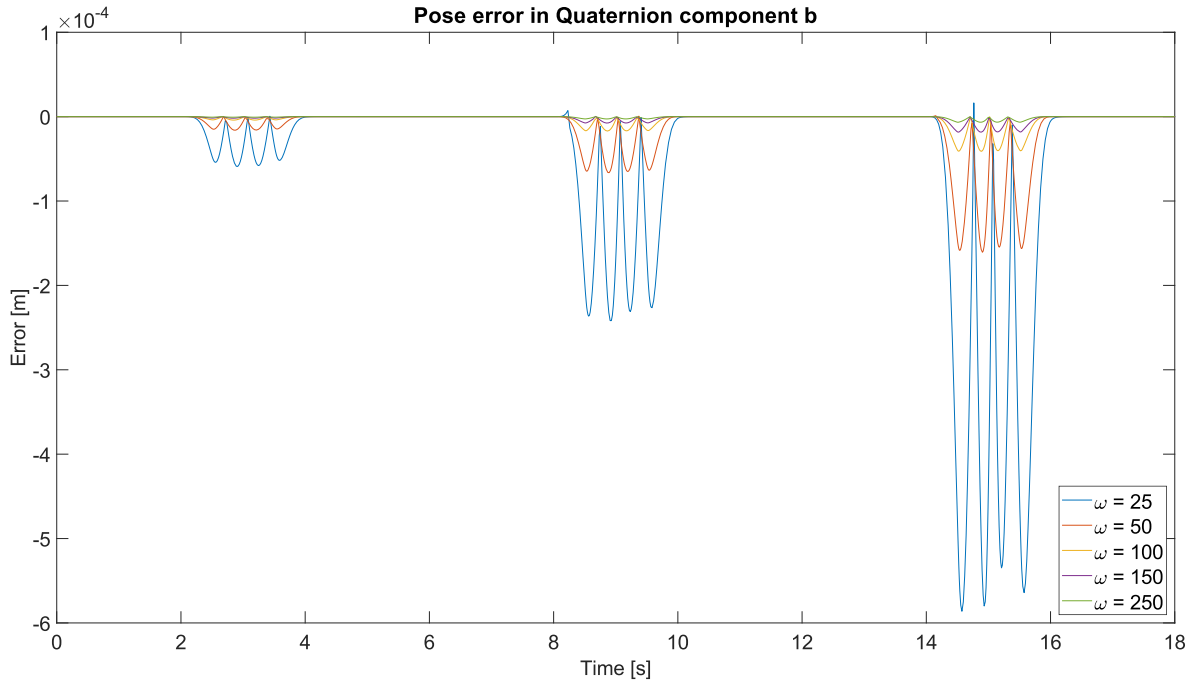

*Plot 3-52. CTC Controller with Trajectory Generator. Committed error in $y$-axis.*



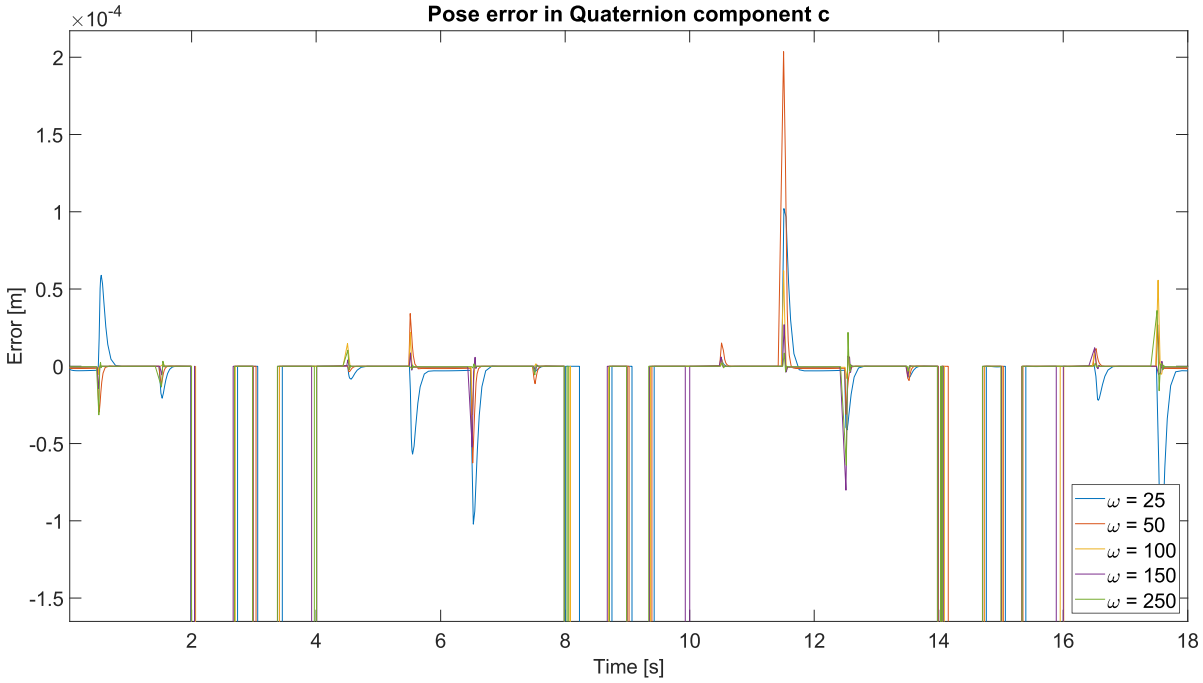

*Plot 3-53. CTC Controller with Trajectory Generator. Committed error in z-axis.*




*Plot 3-54. CTC Controller with Trajectory Generator. Committed error in quaternion component a.*




*Plot 3-55. CTC Controller with Trajectory Generator. Committed error in quaternion component* **b**.




*Plot 3-56. CTC Controller with Trajectory Generator. Committed error in quaternion component* **c**.


*Plot 3-57. CTC Controller with Trajectory Generator. Committed error in quaternion component* **d**.

### 3.5.3 Full system model with a Trajectory Generator block as input and PID Controllers

In this scenario, the Computed Torque Controller was substituted by the PID controllers. Their only input is the vector of joint angles, being the joint velocities and accelerations not necessary.

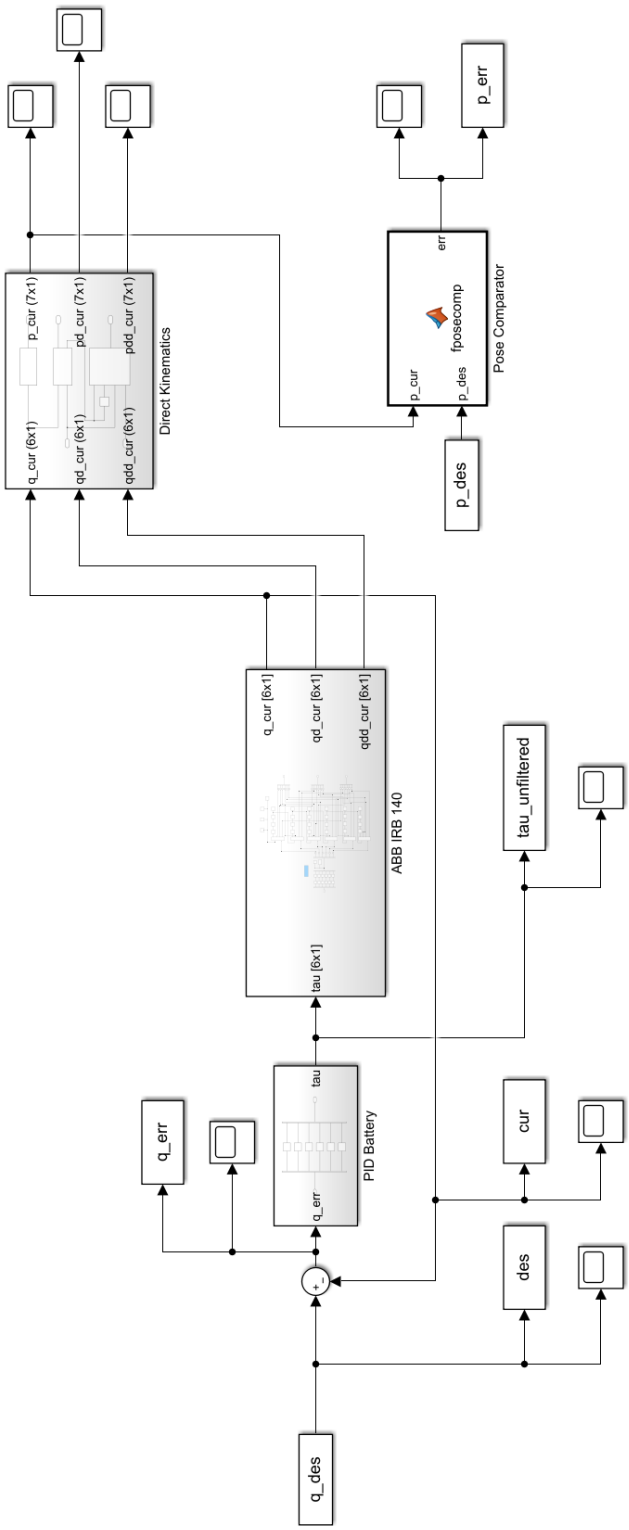
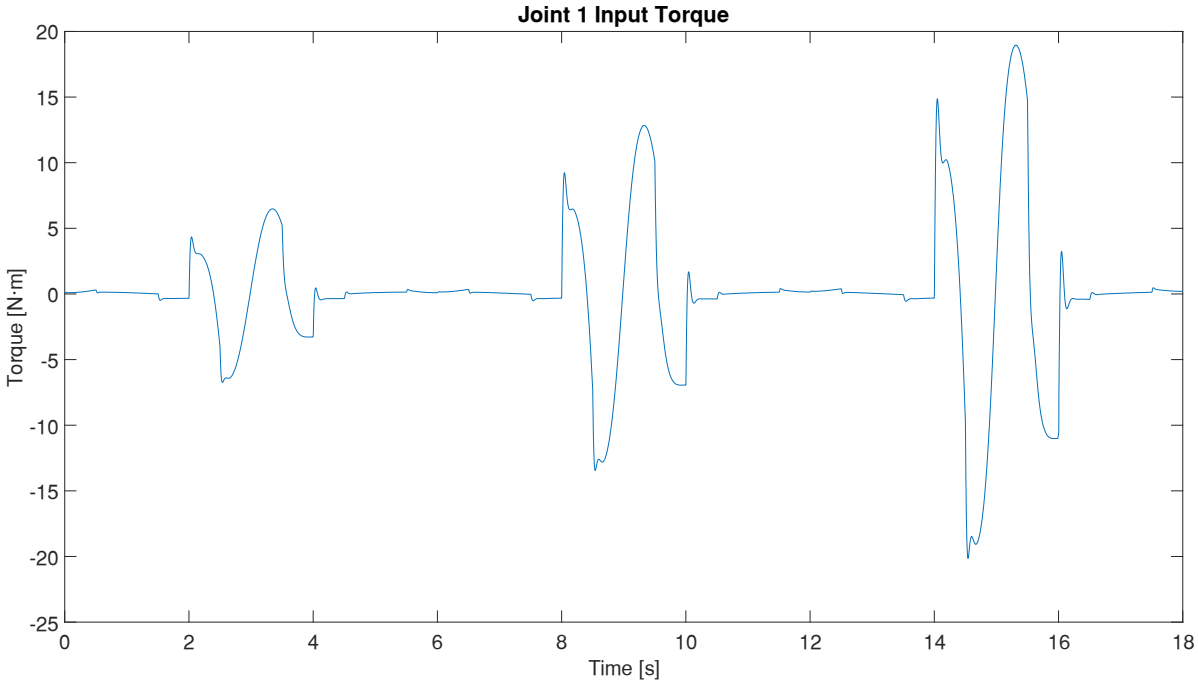


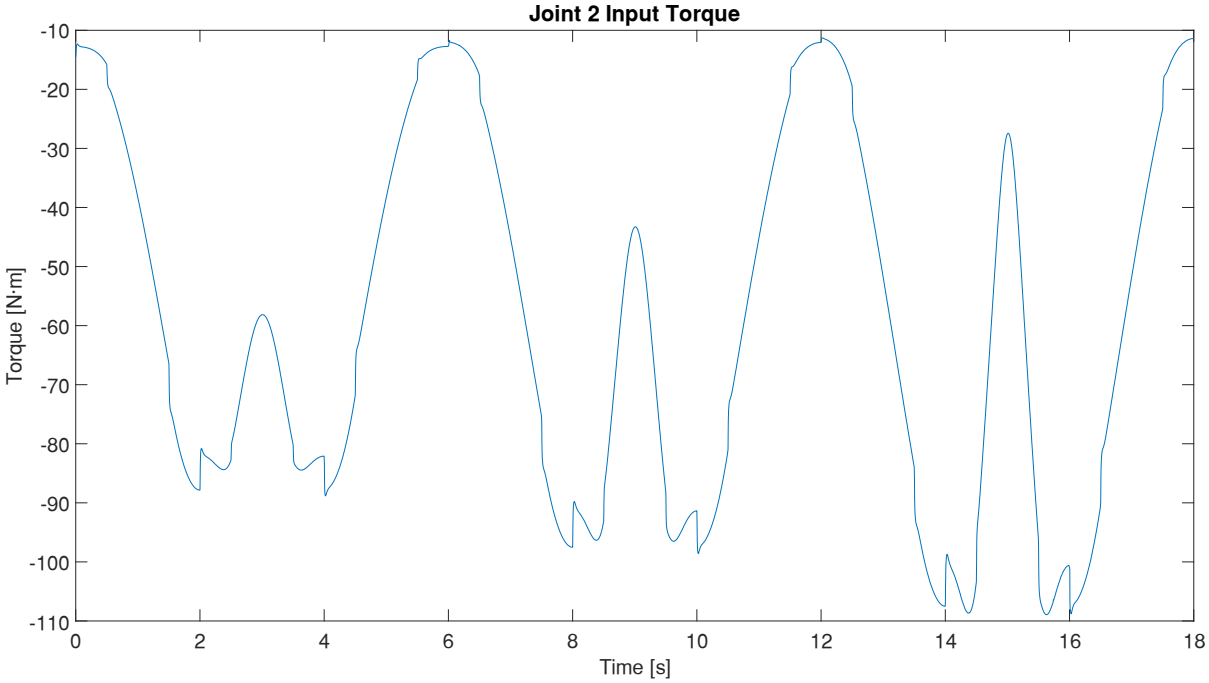

*Figure 3-56. Reduced system schematic with PID Controllers and Trajectory Generator joint angles*

Running this schematic, the results are presented in the next sections.

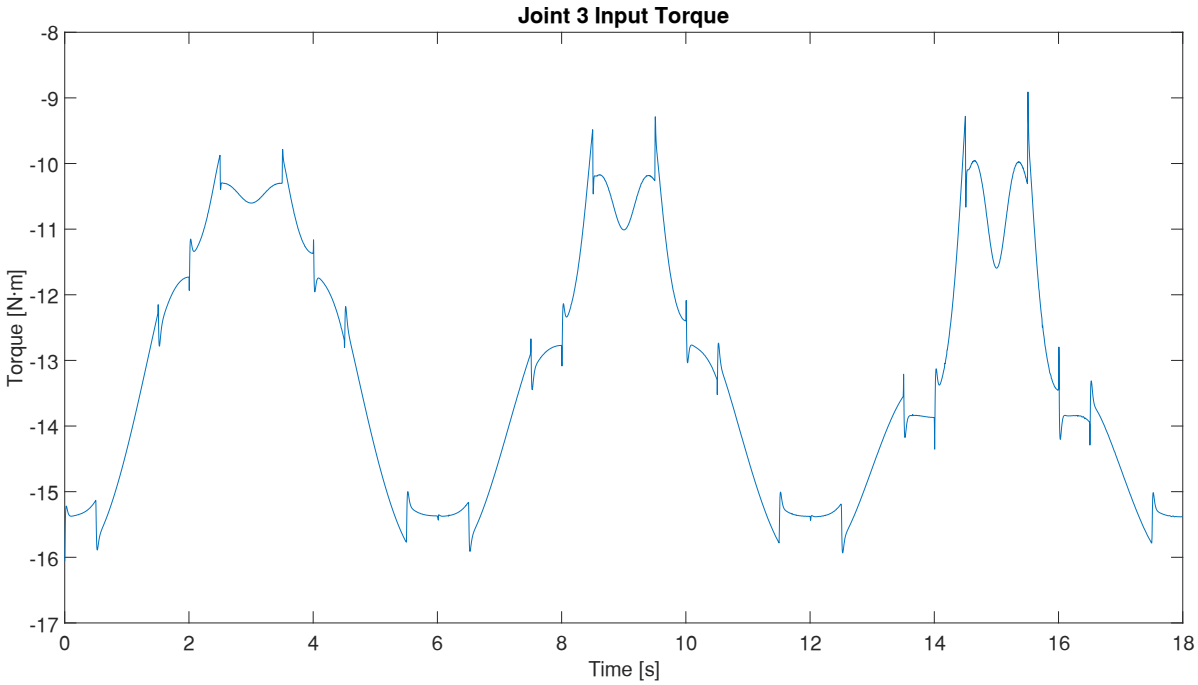
#### *3.5.3.1 Input torques*



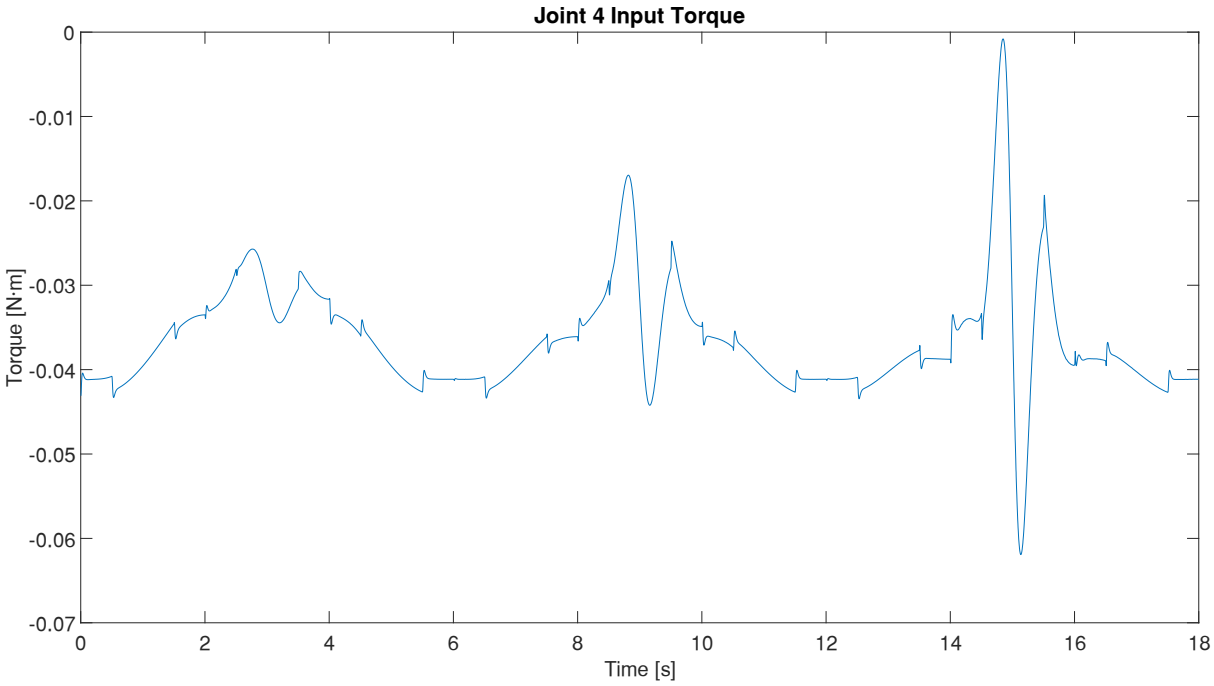

*Plot 3-58. PID Controllers with Trajectory Generator. Input torque in joint 1.*



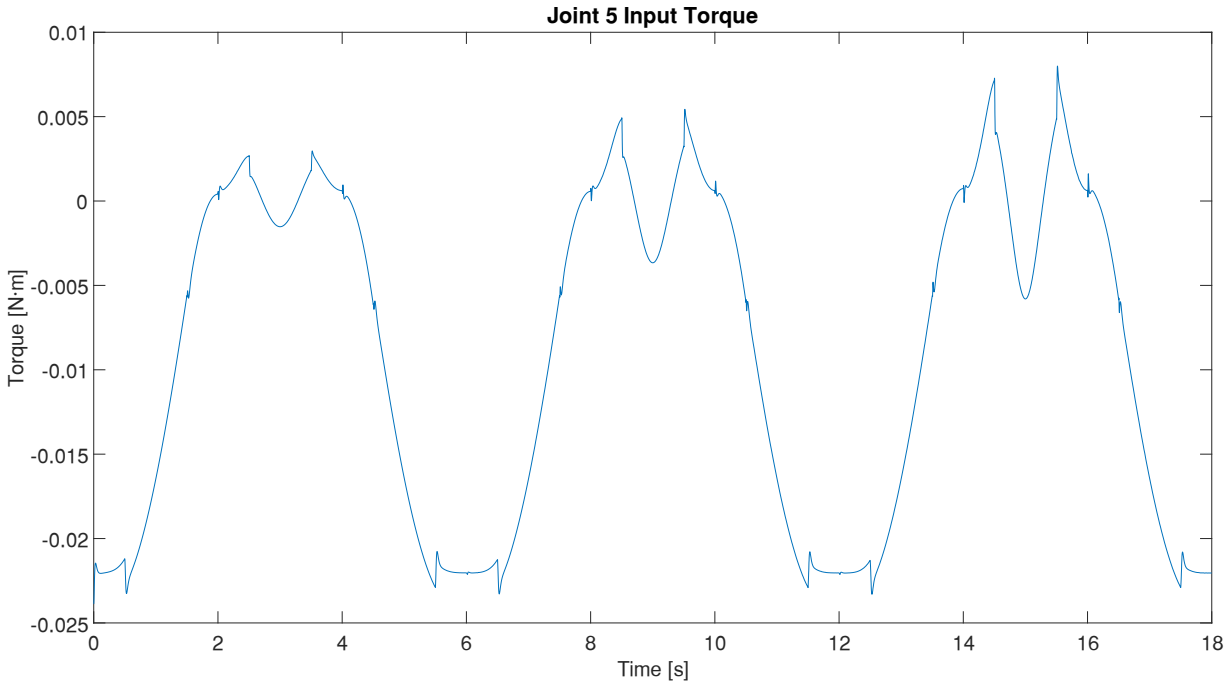

*Plot 3-59. PID Controllers with Trajectory Generator. Input torque in joint 2.*



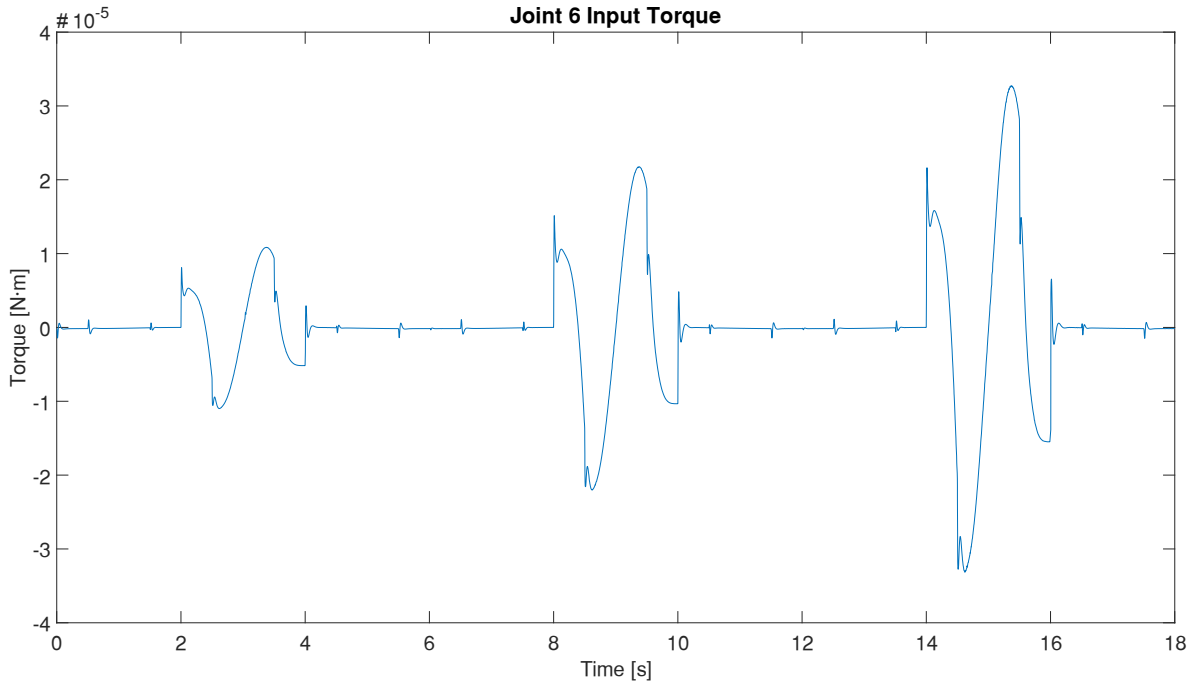

*Plot 3-60. PID Controllers with Trajectory Generator. Input torque in joint 3.*




*Plot 3-61. PID Controllers with Trajectory Generator. Input torque in joint 4.*

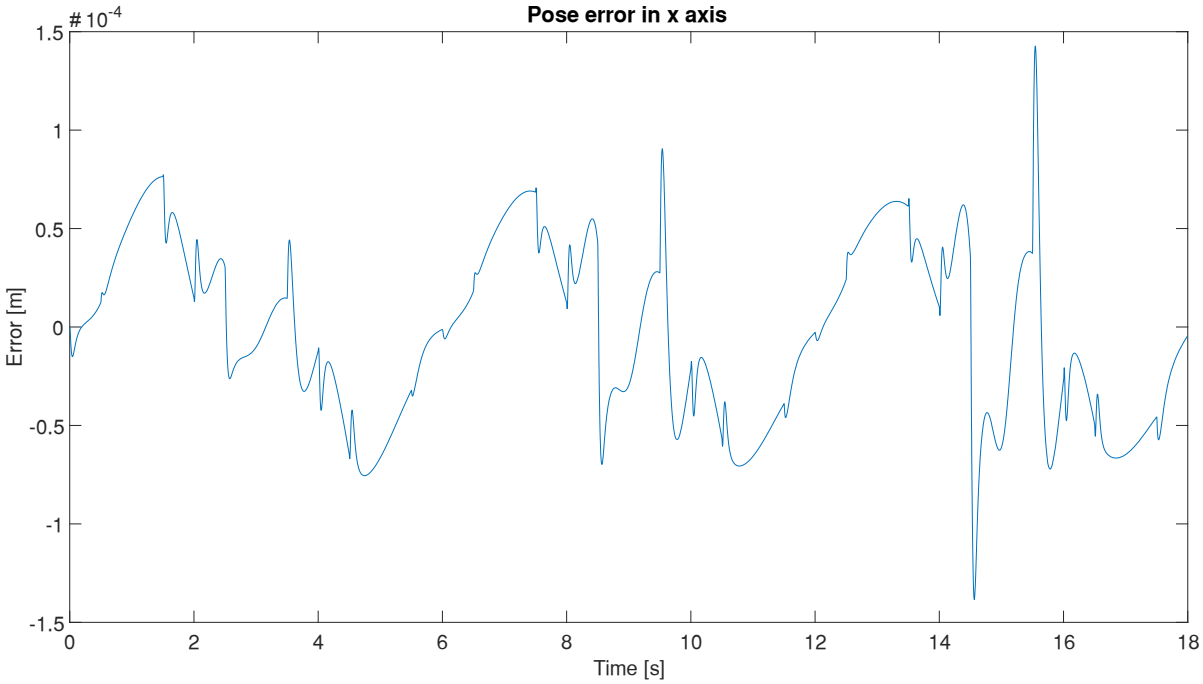



*Plot 3-62. PID Controllers with Trajectory Generator. Input torque in joint 5.*

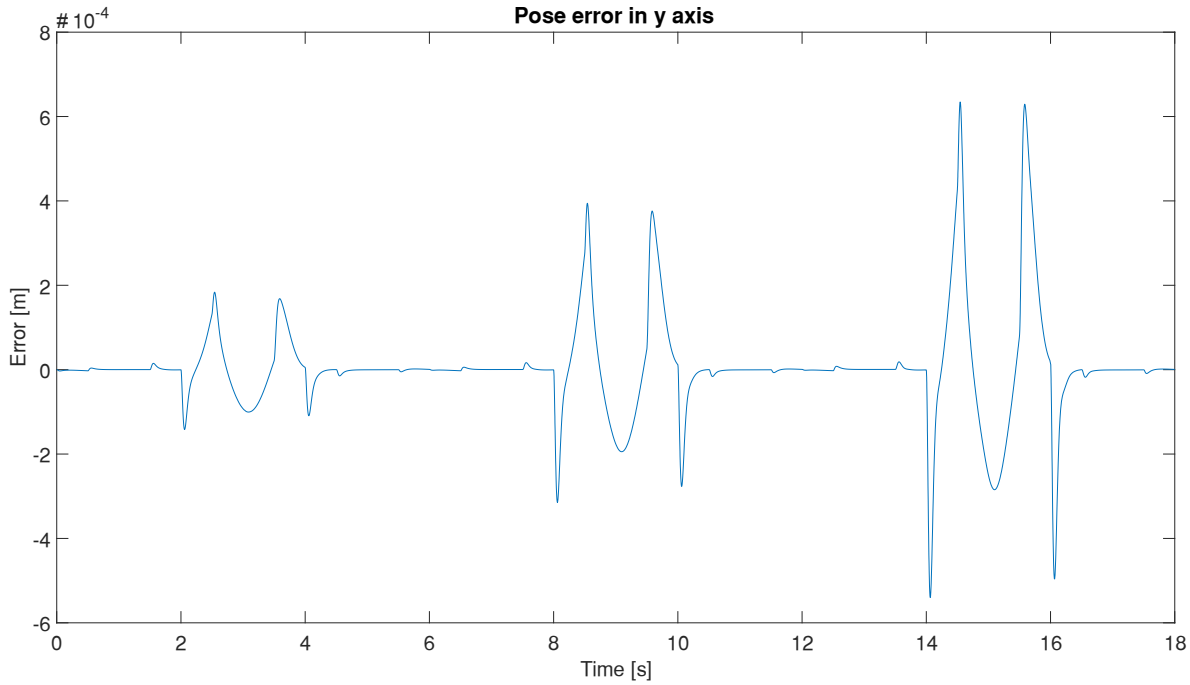



*Plot 3-63. PID Controllers with Trajectory Generator. Input torque in joint 6.*

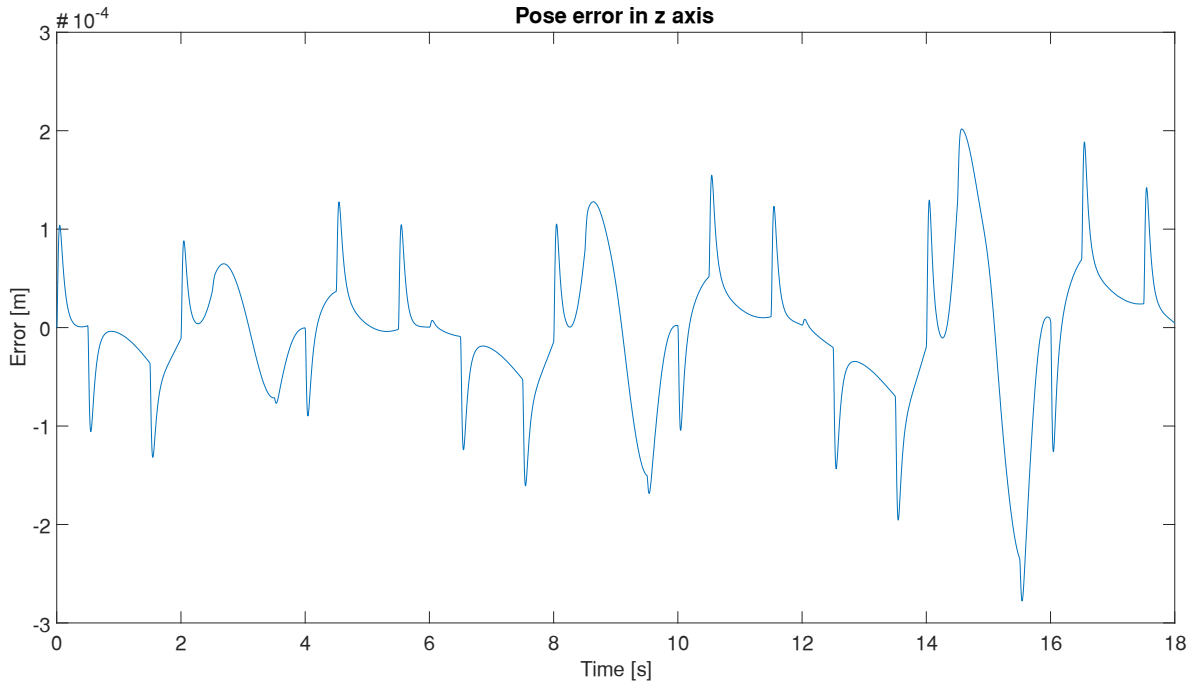
### *3.5.3.2 Pose errors*



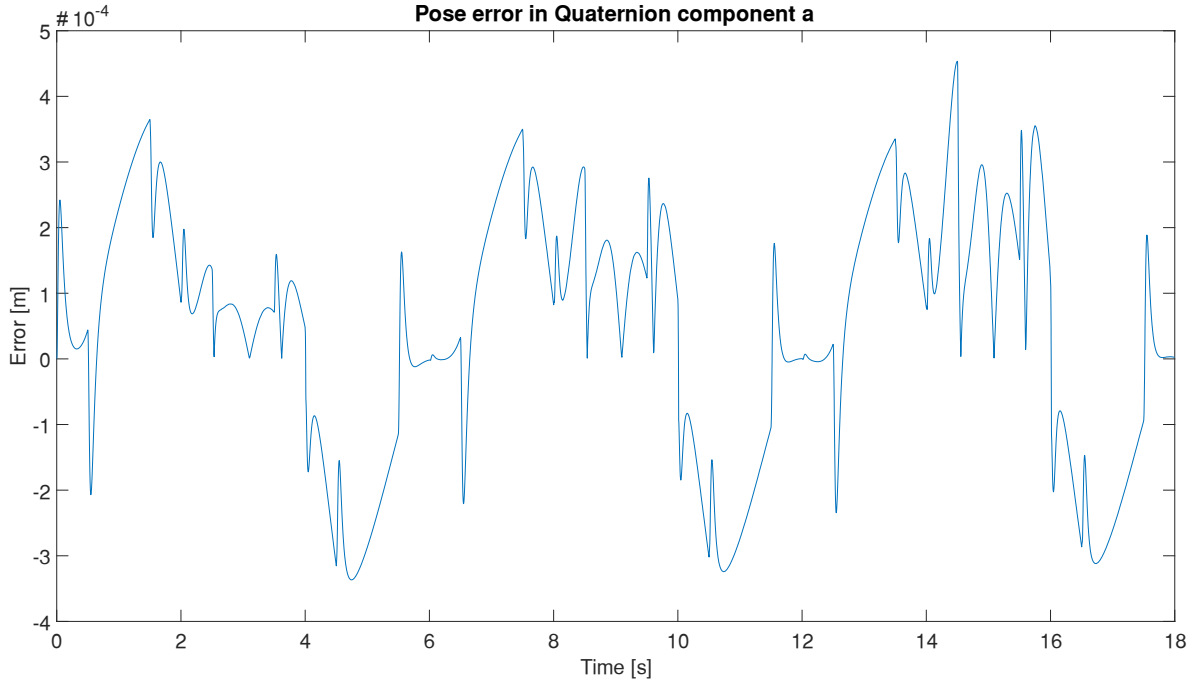

*Plot 3-64. PID Controllers with Trajectory Generator. Committed error in $x$-axis.*



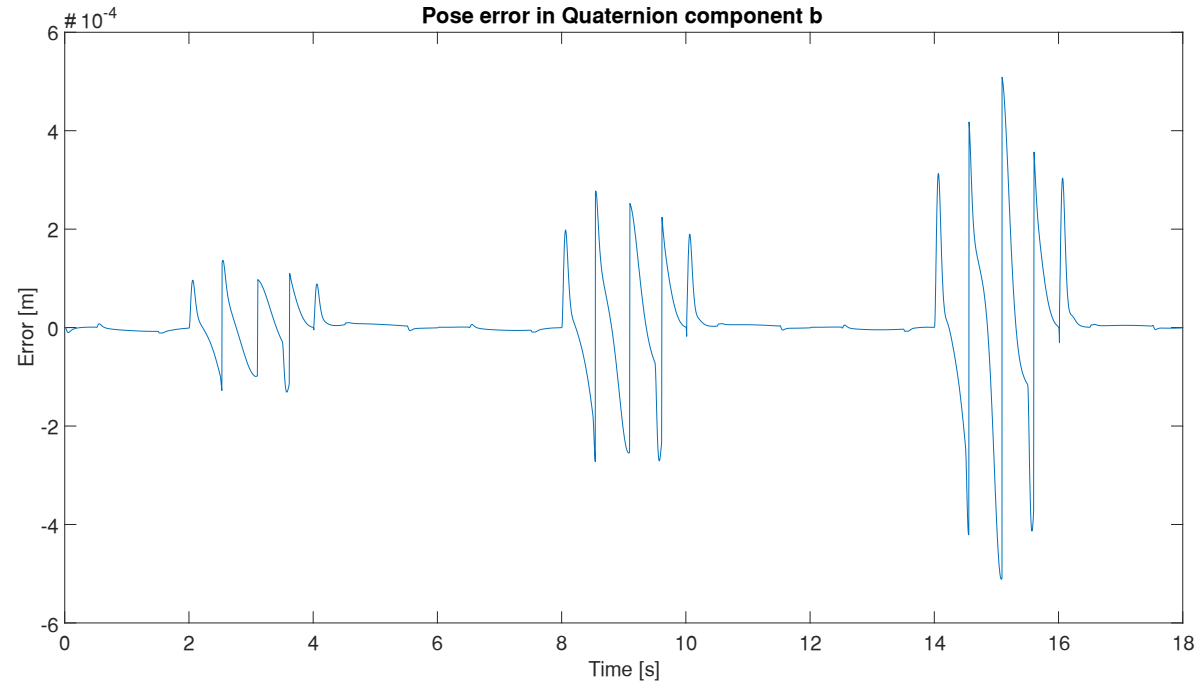

*Plot 3-65. PID Controllers with Trajectory Generator. Committed error in $y$-axis.*



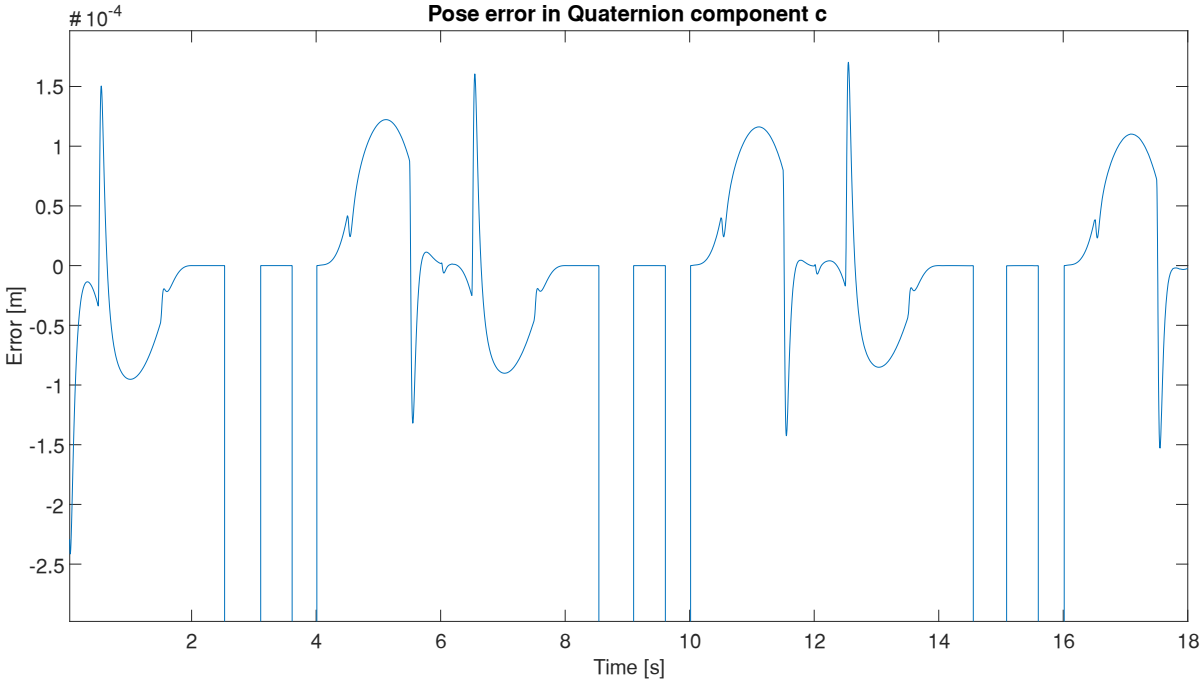

*Plot 3-66. PID Controllers with Trajectory Generator. Committed error in z-axis.*



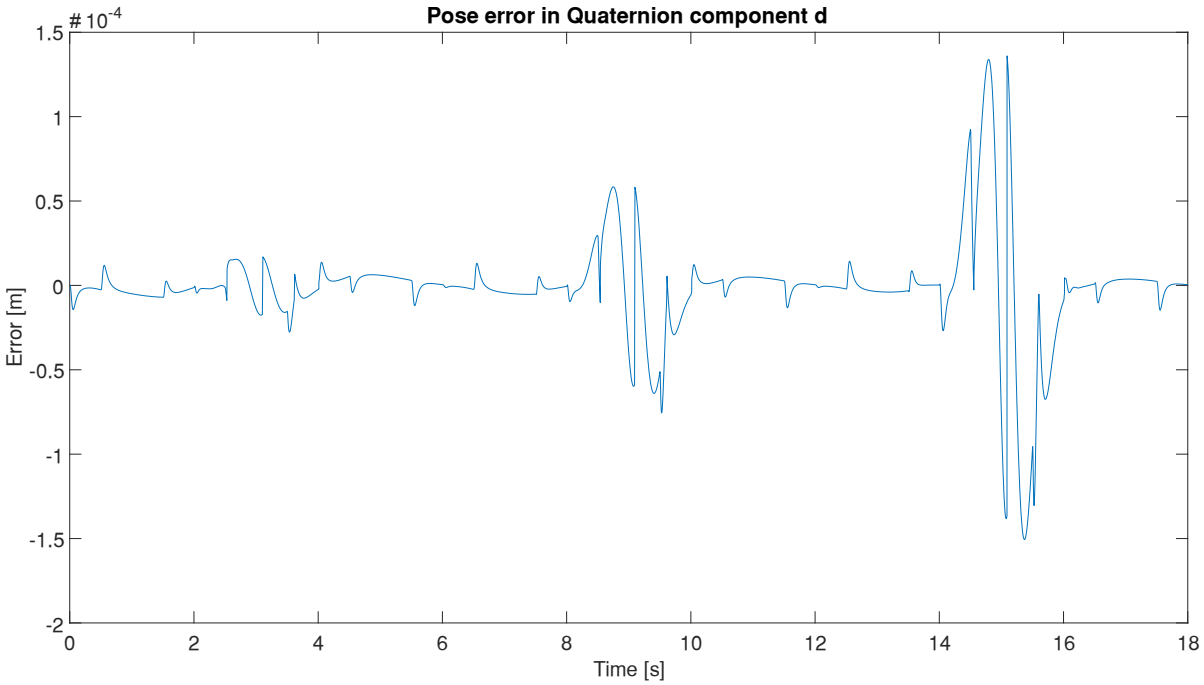

*Plot 3-67. PID Controllers with Trajectory Generator. Committed error in quaternion component a.*




*Plot 3-68. PID Controllers with Trajectory Generator. Committed error in quaternion component* $b$.




*Plot 3-69. PID Controllers with Trajectory Generator. Committed error in quaternion component* $c$.




*Plot 3-70. PID Controllers with Trajectory Generator. Committed error in quaternion component* $d$.

# Chapter 4. Conclusions and future work

This chapter discusses the results of implementation and simulation of all sections of this master's thesis and what could be improved, added or changed in future works.

## 4.1 Conclusions

Although a broadly known pre-built robot software is available on MATLAB (Robotics System Toolbox), it was decided to design all code from scratch to achieve a better understanding of the physical sense of robotics. This way the programmed functions, in contrast with those present in prebuilt libraries, provide the symbolic expressions that regulate the kinematic and dynamic behavior of the ABB IRB 140. Expressions that are needed in order to design a Computed Torque Controller or a PID battery, between many other control solutions.

And this is the major contribution of this master's thesis, the broad analysis performed on the basic fields of robotics such as robot kinematics, robot dynamics and robot control. The material presented here is expected to become useful tools and a reference for future students and researchers.

The individual results of each section (kinematics, dynamics and control) are debated separately down below.

### 4.1.1 Robot kinematics

A well-functioning direct kinematic block was built and implemented. The tests results display a minuscule error in comparison to the RobotStudio configurations extracted for testing, in the order of 0.01~0.1 mm. This same error is later reproduced exactly in the SimScape model testing, which leads to think that a very slightly different method was used to develop the ABB IRB 140 RobotStudio model.

The placement of a hypothetical tool at the end effector such as a pen, has been included with the variable for tool length `Lh`, which displaces the final coordinate system said distance along axis $z$. It is possible that the geometry of the tool gets more complicated, in which case the last transform should be particularly studied.

On the other hand, the inverse kinematic block for poses resulted to require a much more intricated design. In a relatively short time span (~0.1 s) the code designed for this work could find a suitable solution for the problem, although it cannot have into consideration if that reached solution is the optimal. It is most of the time, but the function cannot check it. There is only room to vary the starting iteration vector. Tests were successful but some offered different configuration solutions than the intricated configurations generated in the first place.

The starting vector was first taken from the current vector of joint angles. However, for some reason while staying at singular poses such as the Home pose, where the effects of joints 4 and 6 overlap, the obtained vector of desired joint angles was continuously shifting into different but equally valid joint angles.

Later this was solved when the calculation of joint angles from the pose trajectory was detached from the rest of the model and calculated in advance. The insertion of a column vector of zeros, given the absence of the robot to get the current joint angles from, seemed to solve this problem.

The Jacobian function required some special processing of the symbolic expressions present in the homogeneous transform matrix. The part of the matrix that contains the rotation matrix had to be converted into a quaternion, however there are two different possible symbolic expressions for each quaternion component, whose selection depends on the numeric value of a threshold itself dependent of the rotation matrix values. Because of this, joint angles are needed in order to calculate the proper form of every quaternion component.

Facing the fact that they are not known forehand, all expressions must be directly declared on the function, and let an if/else structure decide which one to use after the joint angles are known.

The present `sign` functions are hard to manipulate and derive. They have all been extracted from the Sarabandi-Thomas method function and grafted after the derivation (the results are equivalent). The resulting Jacobian matrix expressions are then pasted row by row on its correspondent functions.

Once obtained the Jacobian matrix, the behavior of the inverse kinematics blocks for velocities and accelerations was intermittently successful. In certain situations, they acted unexpectedly and in others perfectly fine. This shows that handling large symbolic expressions is prone to errors and inconsistencies. As a result, an interesting different approach to this problem that avoids hard long implementation of these functions and calculation of Jacobian expressions was attempted: numerically deriving joint angles in order to obtain joint velocities and joint accelerations.

A controlled precalculation of these values outperforms that of the Jacobian functions. It makes all unwanted behaviors disappear forming smooth vectors, with customizable time steps

unlike the simulations that include the SimScape robot model. The outcome of this procedure was the final joint coordinate vectors that were called in the definitive simulations.

### 4.1.2 Robot dynamics

The design of the ABB IRB 140 SimScape model ran smoothly and without major surprises. All tests passed and matched with the results of the direct kinematic scripts. Even the inertia matrices calculated by the File Solid blocks were equal to those calculated by SolidWorks.

After the 2-link robot test passed, its code was adapted to fit the geometrical features of the ABB IRB 140, along with all inertias, masses and internal distances. The outcoming matrices M, C and G were tried to be simplified through the MATLAB function `simplify()`, but in the case of matrix C, it took much more time than expected and it was decided to leave it unprocessed.

Due to the still large size of these matrices, the compilation of the function inside a MATLAB function block required around 1 hour of compilation, which means that any small change would demand the whole process to start over. This was the reason the function was implemented in an Interpreted MATLAB function block, in exchange of losing some computing power.

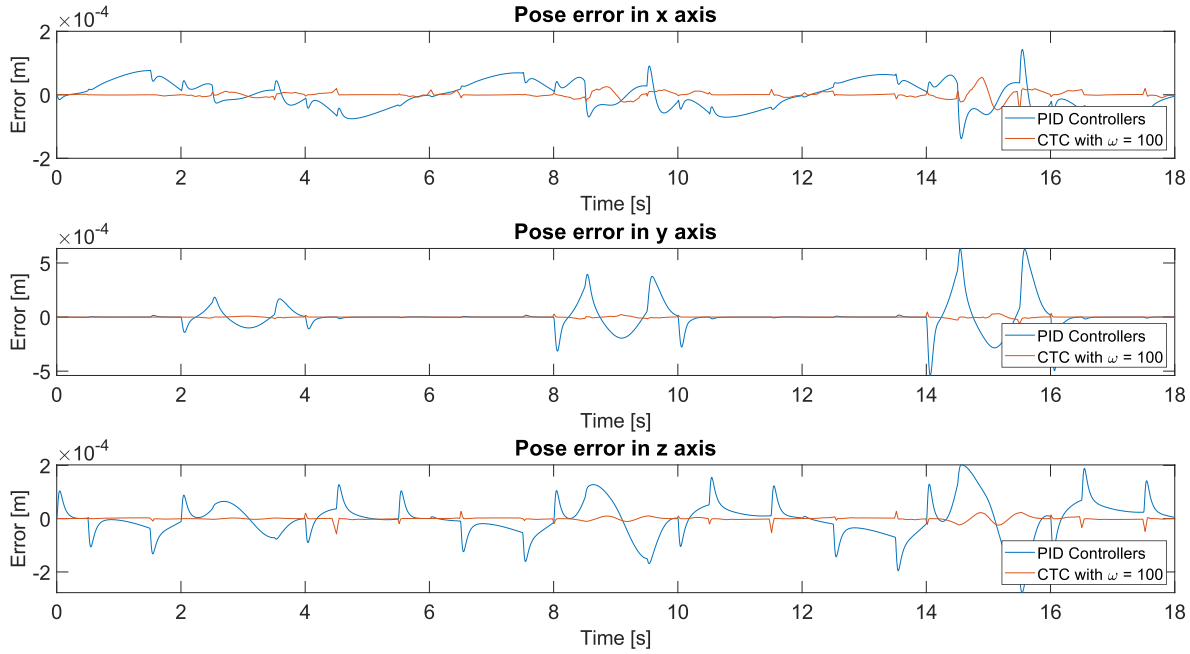
### 4.1.3 Trajectory control

The generation of pose trajectories has proven to be an expandable field with plenty of room for improvement. A simple testing trajectory was developed for the sake of developing a real-life-like continuous trajectory in which all joints could take part.

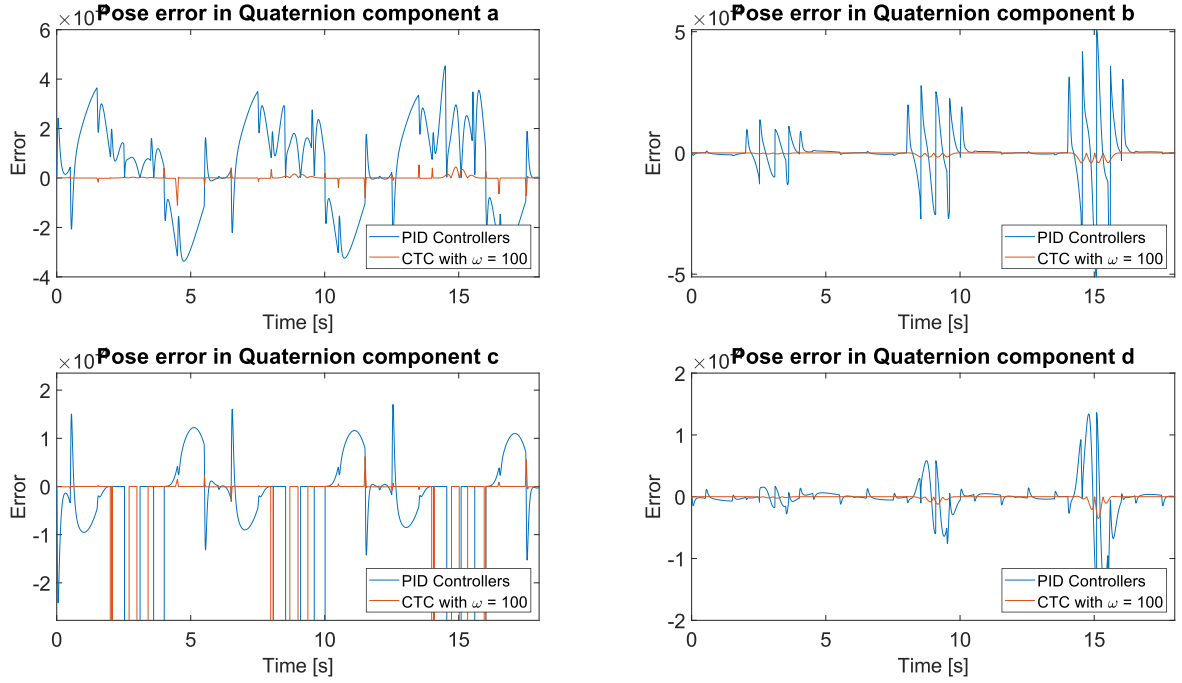
The specific features of actuators ought to be studied in order to draw optimized acceleration profiles that wring the best of the robot capabilities. Arbitrary limits of saturation and slew rate were added but as an approximation. It is the actuator potential what ultimately defines how fine the robot system can be controlled.

Considering that, the data that RobotStudio is able to extract from its simulation is not accurate nor extensive enough, the resulting torques necessary to replicate it turned out oscillatory and not implementable. That is why the comparison between both control laws (CTC and PIDs) have been performed on the trajectory generator inputs. When the CTC bandwidth $\omega$ is 100, it offers a reasonably smooth torque signal with very low errors, therefore the CTC controller with this specific value of bandwidth has been selected.

Merging same pose errors in one plot, a clear understanding of the controller's performance in each specific pose component is accomplished:

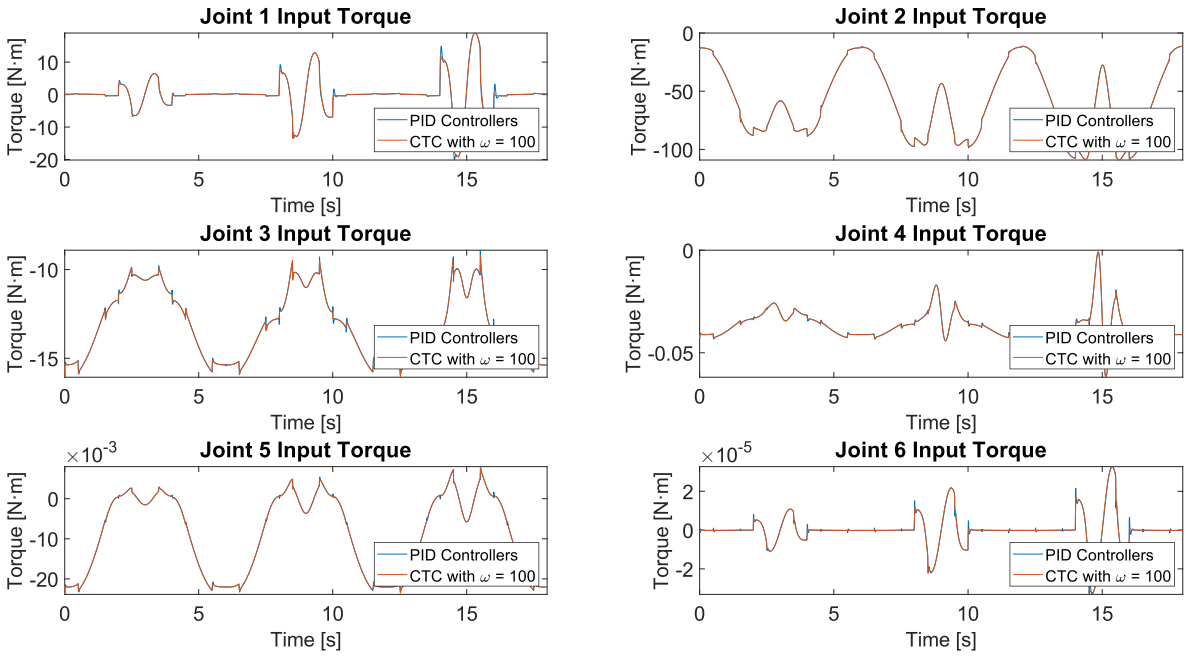



*Plot 4-1. Comparison between PID Controllers and Computed Torque Controllers. Pose error in all axis.*



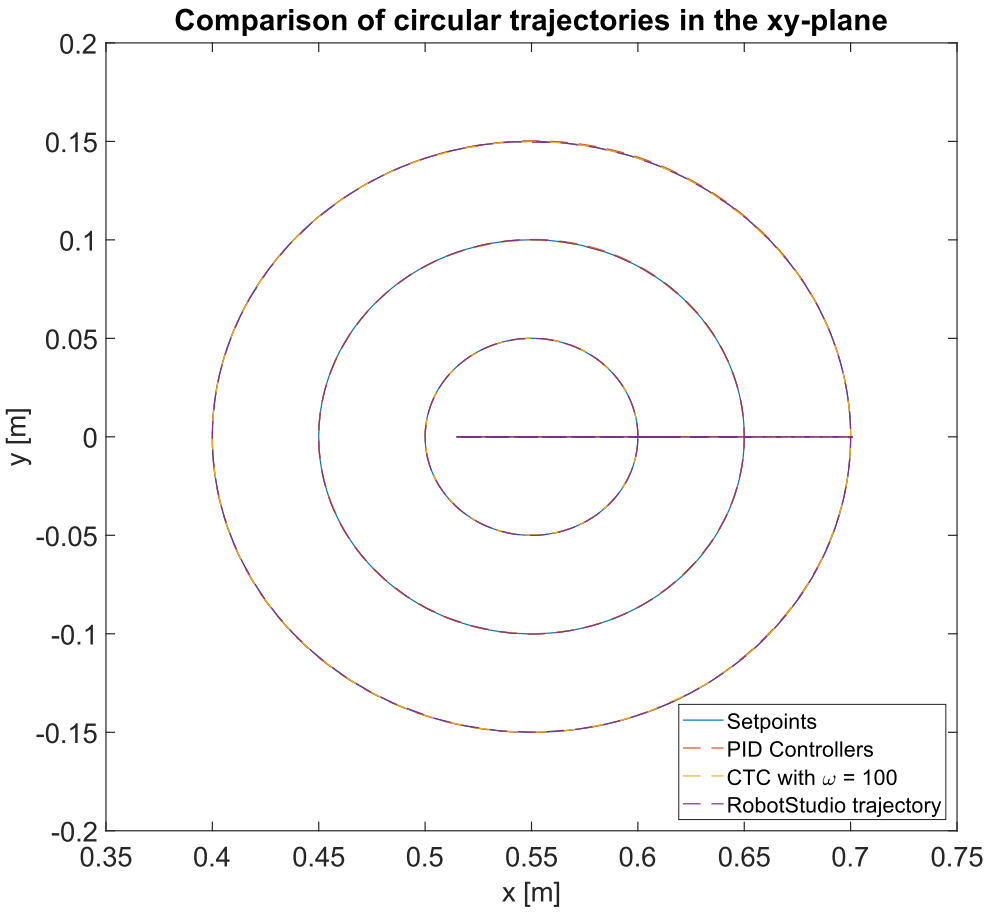

*Plot 4-2. Comparison between PID Controllers and Computed Torque Controllers. Pose error in all quaternion components.*

And comparing torques:

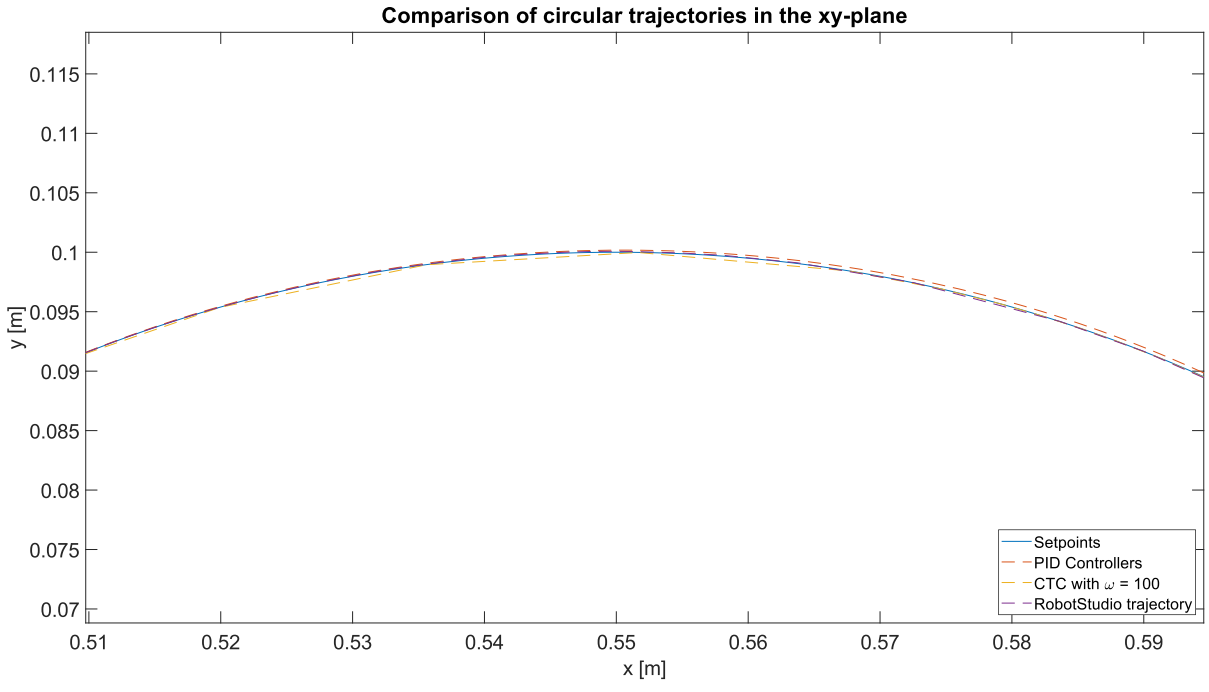



*Plot 4-3. Comparison between PID Controllers and Computed Torque Controllers.*

A plan view of the circles present in the test trajectory is provided, to help this way with a clear scope of the performance of each control solution plus the RobotStudio results, given that the representation is not time dependent.




*Plot 4-4. Comparison between PID Controllers, CTC and RobotStudio. Circular trajectories as seen from above (plane xy).*




*Plot 4-5. Comparison between PID Controllers, CTC and RobotStudio. Circular trajectories closeup.*

According to the comparisons, the best performance is achieved by the Computed Torque Controller, due to a very close equivalence accomplished between the dynamic model and the SimScape model. This is a fantastic example that an accurate design of the dynamic model of a robot can lead to outstanding control solutions with errors in most of the design bandwidths moving in the range of micrometers.

The fact that all CTC errors do not tend to exactly zero over time means that there might be after all slight inaccuracies in the dynamic model or in the simulation configuration itself. This is not ideal, although it is important to highlight that its performance is still excellent. And that is actually the reason the design matrices $K_p$ and $K_v$ are present in the control law, to correct discrepancies between the dynamic model and the dynamic behavior of the robot system.

These discrepancies could come from sources as: the rounding of the huge expressions contained in the M, C and G matrices, some small imprecision in the Newton-Euler algorithm or any other error originating from the very nature of the simulation and its constituent parts.

Any hypothetical implementation of a Computed Torque Controller in a real robot should study thoroughly its real dynamic behavior and reflect it in a model. Phenomena like friction, overheating or braking among other things play a decisive role in the robot dynamics and this control law is so model-dependent that these aspects must indispensably be meticulously designed, especially in case of seeking accurate tracking and positioning. In the case of the existence of loads at the end effector, the code is also prepared to handle them. Only orientation of application of this force should be added along with the value in newtons.

Regarding the PIDs, their aggressive design could cope with the testing trajectory requirements even subjected to torque limitations. Although some overshoot stands out in the torque plots, it is interesting how a quick PID design led to relatively close results compared to a CTC even after many more hours of design and coding. This, together with a greater ease of implementation in real systems and computational robustness, is a clear sign of how powerful the PID controllers are as control solutions. And PID controllers are actually used in most of modern industrial robots.

In the plot of the plan view of the circles, it was also notable that the CTC results were slightly improving the RobotStudio results. Those instants in which the CTC results got further away from the setpoints were product of the different time steps of the simulations.

A useful application of power consumption calculation is that it allows the electrical dimensioning of the manufacturing cell the robot is going to be part of. Along with a safety coefficient, this modelling process could offer a decent approximation of the electrical requirements of any robot.

## 4.2 Further work

As it was pointed out previously, this master's thesis extends over many fields of robotics and these are rich enough to provide many alternative solutions and/or expansions to all problems addressed here. Some proposed are:

### 4.2.1 Improvement of the inverse kinematics function

A different function than `fsolve` would be desirable. It would have to be written fully from scratch to consider angle and angular velocity limitations, apart from taking into consideration all possible robot configurations that can reach a single pose. It is worth mentioning that the Robotics System Toolbox has already one specific inverse kinematics function.

### 4.2.2 Refinement of the dynamic model

As said in section 4.1.3, new tests could be built and ran, to check where the inaccuracies come from and correct the Newton-Euler code. This improvement, in the case of a real robot, could be extended to cover new aspects of robot dynamics as for example, friction. There is plenty of ongoing research about dynamic modeling.

### 4.2.3 Use of a physical robot for parameter identification

The actual current passed onto the actuators can be measured through current clamps and used as another resource to obtain an energy-efficient dynamic model of the robot, as performed in [12].

Data could be extracted from the robot system, specifically joint angles, velocities and accelerations, and end effector poses. Combinations of these would be the references in modeling through Least Square techniques.

[13] is an example of the above. It presents a matrix form of simplified Lagrangian equations for dynamic modelling and through Least Square techniques and verification trajectories, finds the most suitable dynamic parameters that can accomplish these trajectories.

### 4.2.4 CTC and PID parameter optimization

CTC parameters $K_p$ and $K_v$ can be optimized iteratively by testing thousands of combinations and keeping the ones that cause the least errors. Similar way with PID controllers and their parameters $K_p$, $K_i$ and $K_d$. Stalling simulations should be monitored and terminated.

### 4.2.5 CTC initialization

An interesting concept would be to initialize the controller with the torques necessary to keep the robot at home position, as it was done with the PID controllers to avoid initial torque overshoots.

# Chapter 5. Bibliography


[1] ABB Product specification, ABB IRB 140.

[2] Denavit, J. and Hartenberg, R.S. (1955). A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices. *ASME Journal of Applied Mechanics*, 77, 215-221.

[3] Almaged, M. (2017). Forward and Inverse Kinematic Analysis and Validation of the ABB IRB 140 Industrial Robot. *International Journal of Electronics, Mechanical and Mechatronics Engineering*, *7*(2), 1383–1401. http://doi.org/10.17932/iau.ijemme.21460604.2017.7/2.1383-1401

[4] Suárez Baquero, M. & Ramírez Heredia, R. (2015). Kinematics, Dynamics and Evaluation of Energy Consumption for ABB IRB-140 Serial Robots in the Tracking of a Path Kinematics, Dynamics and Evaluation of Energy Consumption for ABB IRB-140 Serial Robots in the Tracking of a Path (October 2013). http://doi.org/10.13140/2.1.3436.5448

[5] Córdoba López, A. J. (2016). Puesta en marcha de brazo robótico y desarrollo de aplicaciones. *Trabajo Fin de Grado. Escuela Politécnica Superior de Linares.*

[6] Mato San José, M. A. (2014). Simulación, control cinemático y dinámico de robots comerciales usando la herramienta de MATLAB Robotic Toolbox. *Trabajo Fin de Grado. Universidad de Valladolid. Escuela de Ingenierías Industriales.*

[7] Rønnestad, R. A. (2013). Scenario Development for Industrial Robot Manipulator: Case Study in Processing Casted Metal Components. *Master's Thesis. Norwegian University for Science and Technology.*

[8] Barrientos, A. (2007). Fundamentos de Robótica. (McGraw-Hill Interamericana de España S.L, Ed.) (2º).

[9] Corke, P. (2017). Robotics, Vision and Control: Fundamental Algorithms in MATLAB. (Springer, Ed.) (1º). http://doi.org/3642201431

[10] Kelly, R, & Santibáñez, V. (2013). Control de Movimiento de Robots Manipuladores. (Pearson Educación)

[11] Sarabandi, S., & Thomas, F. (n.d.). Accurate Computation of Quaternions from Rotation Matrices. *Institut de Robòtica i Informàtica Industrial (CSIC-UPC)*

[12] Paes, K., Dewulf, W., Vander, K., Kellens, K., & Slaets, P. (2014). Energy efficient trajectories for an industrial ABB robot. *21st CIRP Conference on Life Cycle Engineering* 15, 105–110. https://doi.org/10.1016/j.procir.2014.06.043

[13] Memar, A. H., & Esfahani, E. T. (2015). Modeling and Dynamic Parameter Identification of the SCHUNK. *Proceedings of the ASME 2015 International Design Engineering Technical Conferences & Computer and Information in Engineering Conference*. https://doi.org/10.1115/DETC2015-47703

# Appendix. Code

This appendix gathers the MATLAB and RAPID code used in the master's thesis. Many expressions have been removed and written as comment due to their exceptional size.

## Code 1. Sarabandi-Thomas method for the computation of quaternions from rotation matrices

```
function q = sarabandi_thomas(R)
% Computation of a quaternion q from a rotation matrix R according to
% the Sarabandi-Thomas method

eta = 0;        % Threshold
q = zeros(4,1); % Declares quaternion

if R(1,1)+R(2,2)+R(3,3) > eta
    q(1) = 0.5*sqrt(1+R(1,1)+R(2,2)+R(3,3));
else
    q(1) = 0.5*sqrt(((R(3,2)-R(2,3))^2+(R(1,3)-R(3,1))^2+(R(2,1)-
R(1,2))^2)/(3-R(1,1)-R(2,2)-R(3,3)));
end

if R(1,1)-R(2,2)-R(3,3) > eta
    q(2) = 0.5*sqrt(1+R(1,1)-R(2,2)-R(3,3));
else
    q(2) = 0.5*sqrt(((R(3,2)-
R(2,3))^2+(R(1,2)+R(2,1))^2+(R(3,1)+R(1,3))^2)/(3-R(1,1)+R(2,2)+R(3,3)));
end

if -R(1,1)+R(2,2)-R(3,3) > eta
    q(3) = 0.5*sqrt(1-R(1,1)+R(2,2)-R(3,3));
else
    q(3) = 0.5*sqrt(((R(1,3)-
R(3,1))^2+(R(1,2)+R(2,1))^2+(R(2,3)+R(3,2))^2)/(3+R(1,1)-R(2,2)+R(3,3)));
end
```

```
if -R(1,1)-R(2,2)+R(3,3) > eta
    q(4) = 0.5*sqrt(1-R(1,1)-R(2,2)+R(3,3));
else
    q(4) = 0.5*sqrt(((R(2,1)-
R(1,2))^2+(R(3,1)+R(1,3))^2+(R(3,2)+R(2,3))^2)/(3+R(1,1)+R(2,2)-R(3,3)));
end

q(2) = q(2)*sign(R(3,2)-R(2,3));
q(3) = q(3)*sign(R(1,3)-R(3,1));
q(4) = q(4)*sign(R(2,1)-R(1,2));

end
```

## Code 2. Symbolic Sarabandi-Thomas method for the computation of quaternions from rotation matrices

```
function q = sarabandi_thomas_sym(R)

q = sym('q',[8 1],'real'); % Declares all possible values of quaternion

q(1) = 0.5*sqrt(1+R(1,1)+R(2,2)+R(3,3));
q(2) = 0.5*sqrt(((R(3,2)-R(2,3))^2+(R(1,3)-R(3,1))^2+(R(2,1)-R(1,2))^2)/(3-
R(1,1)-R(2,2)-R(3,3)));

q(3) = 0.5*sqrt(1+R(1,1)-R(2,2)-R(3,3));
q(4) = 0.5*sqrt(((R(3,2)-R(2,3))^2+(R(1,2)+R(2,1))^2+(R(3,1)+R(1,3))^2)/(3-
R(1,1)+R(2,2)+R(3,3)));

q(5) = 0.5*sqrt(1-R(1,1)+R(2,2)-R(3,3));
q(6) = 0.5*sqrt(((R(1,3)-
R(3,1))^2+(R(1,2)+R(2,1))^2+(R(2,3)+R(3,2))^2)/(3+R(1,1)-R(2,2)+R(3,3)));

q(7) = 0.5*sqrt(1-R(1,1)-R(2,2)+R(3,3));
q(8) = 0.5*sqrt(((R(2,1)-
R(1,2))^2+(R(3,1)+R(1,3))^2+(R(3,2)+R(2,3))^2)/(3+R(1,1)+R(2,2)-R(3,3)));

% No multiplication by sign function yet, to avoid overcomplicated
% derivatives

end
```

## Code 3. Obtention of the ABB IRB 140 Homogeneous Transformation Matrix

```
%% Master's Thesis | ABB IRB 140 | Direct Kinematics

q = sym('q',[6 1]); % Symbolic column vector with 6 joint angles
Lh = 0;             % Tool length

% D-H parameters table
DH = [q(1)          0.352    0.070   -pi/2;
      (pi/2)+q(2)   0        -0.360  0;
      q(3)          0        0       pi/2;
      q(4)          0.38     0       -pi/2;
      q(5)          0        0       pi/2;
      q(6)          0.065+Lh 0       0];

[m,n] = size(DH);   % m = number of d.o.f.'s
T = eye(n);

for i = 1:m
    A = [cos(DH(i,1))    -cos(DH(i,4))*sin(DH(i,1))
sin(DH(i,4))*sin(DH(i,1))    DH(i,3)*cos(DH(i,1));
         sin(DH(i,1))    cos(DH(i,4))*cos(DH(i,1))    -
sin(DH(i,4))*cos(DH(i,1))  DH(i,3)*sin(DH(i,1));
         0               sin(DH(i,4))                 cos(DH(i,4))
DH(i,2);
         0               0                            0
1];

    T = T*A;
end

T = simplify(T);            % Simplifies the transformation matrix
```

## Code 4. Direct kinematics function

```
function p_cur = fdirkin(q_cur)
% Obtains the translation vector and quaternion asociated to an input
% transformation matrix. They both form the (current) pose vector.

t = ...
% Long T declaration obtained from Code 1 with q_cur (6x1) as input

p_cur = zeros(7,1);
p_cur(1:3,1) = t(1:3,4);                          % Translation vector
p_cur(4:7,1) = sarabandi_thomas(t(1:3,1:3));      % Quaternion
end
```

## Code 5. Jacobian matrix function

```
%% Master's Thesis | ABB IRB 140 | Jacobian matrix generator

q = sym('q',[6 1]); % Symbolic column vector with 6 joint angles
Lh = 0;             % Tool length

% D-H parameters table
DH = [q(1)        0.352       0.070  -pi/2;
    (pi/2)+q(2)   0          -0.360  0;
    q(3)          0           0      pi/2;
    q(4)          0.380       0      -pi/2;
    q(5)          0           0      pi/2;
    q(6)          0.065+Lh    0      0];

[m,n] = size(DH);   % m = filas, n = columnas
T = eye(n);

for i = 1:m
    A = [cos(DH(i,1))    -cos(DH(i,4))*sin(DH(i,1))
sin(DH(i,4))*sin(DH(i,1))    DH(i,3)*cos(DH(i,1));
        sin(DH(i,1))    cos(DH(i,4))*cos(DH(i,1))    -
sin(DH(i,4))*cos(DH(i,1))  DH(i,3)*sin(DH(i,1));
        0               sin(DH(i,4))                 cos(DH(i,4))
DH(i,2);
        0               0                            0
1];

    T = T*A;
end

T = simplify(T);

transvec = T(1:3,4);     % Extracts translation vector
rotmat = T(1:3,1:3);     % Extracts rotation matrix

symquat = sarabandi_thomas_sym(rotmat); % Creates a symbolic vector of
                                        % 8 variables, 4 quaternion
                                        % componentes times 2 possibilities
                                        % each, to derive them separately

pose = [transvec; symquat]; % Stacks in a vector 3 translation values
                            % and 4x2=8 quaternion posible values. Total 11

J = sym('J',[11 6],'real');
[rows,columns] = size(J);

for i = 1:rows
    for j = 1:columns
        J(i,j) = diff(pose(i),q(j));    % Derives each row by each joint

    end
end

% The sign function is then added
for i = 1:columns
    J(6,i) = J(6,i)*sign(rotmat(3,2)-rotmat(2,3));
    J(7,i) = J(7,i)*sign(rotmat(3,2)-rotmat(2,3));
```

```
    J(8,i) = J(8,i)*sign(rotmat(1,3)-rotmat(3,1));
    J(9,i) = J(9,i)*sign(rotmat(1,3)-rotmat(3,1));
    J(10,i) = J(10,i)*sign(rotmat(2,1)-rotmat(1,2));
    J(11,i) = J(11,i)*sign(rotmat(2,1)-rotmat(1,2));
end
```

## Code 6. Direct kinematics function for velocities

```
function [pd_cur,J] = fdirkinvel(q_cur,qd_cur)
% Calculates the Jacobian matrix from current joint angles and operates
% with joint velocities to find the pose velocity

J = zeros(7,6);

J(1:3,:) = [...
% First 3 rows of the Jacobian matrix

if % Condition above threshold (threshold = 0)
      % First possible 4th row of J (1st quaternion component)
else
      % Second possible 4th row of J
end

if % Condition above threshold
      % First possible 5th row of J (2nd quaternion component)
else
      % Second possible 5th row of J
end

if % Condition above threshold
      % First possible 6th row of J (3rd quaternion component)
else
      % Second possible 6th row of J
end

if % Condition above threshold
      % First possible 7th row of J (4th quaternion component)
else
      % Second possible 7th row of J
end

pd_cur = J*qd_cur;

end
```

## Code 7. Direct kinematics function for accelerations

```
function pdd_cur = fdirkinacc(J,dJdt,qd_cur,qdd_cur)
% Calls the current value of the Jacobian matrix and its derivative,
% operates with them and the current joint accelerations and velocities
% to obtain the pose acceleration

pdd_cur = J*qdd_cur + dJdt*qd_cur;
end
```

## Code 8. Pose comparator function

```
function equal = fcomp(p1,p2)
% Compares 2 vectors from 2 consecutive iterations, and if they're equal
% it returns 1, if not, 0. Boolean values aren't used as if blocks don't
% accept them.

equal = 1;
c = 1;

% Until the loop doesn't find 2 different values or exceeds vector length
% it keeps running
while equal == 1 && c <= length(p1)
    if p1(c,1) ~= p2(c,1)
        equal = 0;
    end
    c = c+1;
end
end
```

## Code 9. Inverse kinematics function

```
function [q_des] = finvkin(p_qcur)
% Obtains joint coordinates q, necessary to reach a certain end effector
% pose. Simulink MATLAB function blocks can't handle fsolve, so this
% function is called by an Interpreted MATLAB function block.
% Interpreted MATLAB function blocks can't handle more than 1 input, that
% is the desired pose and current joint coordinates are muxed and inserted
% as a single vector.

rotmat = quat2rotm(p_qcur(4:7)');    % Converts the quaternion from the pose
                                     % into a rotation matrix
Tobj = [rotmat p_qcur(1:3)];     % Creates a transformation matrix with
                                 % only 3 rows to gather all objective
                                 % values (rotation matrix & translation)

b = zeros(12,1);     % A column vector of independent terms is declared

% These objective values are saved in the b vector one by one
for i = 1:3
    for j = 1:4
        b(4*(i-1)+j,1) = Tobj(i,j);
    end
end

exitflag = -2;  % States the solver exit flag as "failed" to initialize
                % the while loop
c = 0;       % Counter to know how many times there has been an attempt
                % to solv the non-linear equation system

% exitflag > 0 -> solution found | <= 0 -> solution not found
% While loop tries 20 times increasing the initial value by 1. After these
% if there's no solution, quits the loop.

while exitflag <= 0 || c == 20
    [q_des,fval,exitflag] = fsolve(@(q) fecnolin(q,b),p_qcur(8:13)');
    if exitflag ~= 1
        c = c+1;
        p_qcur(8:end) = p_qcur(8:end)+1;
    end
end
q_des = q_des';
end
```

## Code 10. Inverse kinematics function for velocities

```
function [qd_des,J] = finvkinvel(q_des,pd_des)

J = zeros(7,6);

J(1:3,:) = [...
% Long declaration of the 3 first J rows

if % Condition above threshold (threshold = 0)
      % First possible 4th row of J (1st quaternion component)
else
      % Second possible 4th row of J
end

if % Condition above threshold
      % First possible 5th row of J (2nd quaternion component)
else
      % Second possible 5th row of J
end

if % Condition above threshold
      % First possible 6th row of J (3rd quaternion component)
else
      % Second possible 6th row of J
end

if % Condition above threshold
      % First possible 7th row of J (4th quaternion component)
else
      % Second possible 7th row of J
end

% Note: sign functions have been substituted by a modified version of it
% where if the input is 0, it returns 1. This avoid ill-conditioned
% situations where the algorithm would crash

qd_des = J\pd_des;

end
```

## Code 11. Inverse kinematics function for accelerations

```
function qdd_des = finvkinacc(qd_des,J,dJdt,pdd_des)

qdd_des = J\(pdd_des-dJdt*qd_des);

end
```

## Code 12. Newton-Euler algorithm for the obtention of the ABB IRB 140 dynamic model

```
%% Master's Thesis | ABB IRB 140 | Newton-Euler Algorithm

clear all
close all
clc

q = sym('q',[6 1],'real');      % 6 joint angles
qd = sym('qd',[6 1],'real');    % 6 joint velocities
qdd = sym('qdd',[6 1],'real'); % 6 joint accelerations
Lh = 0;                         % Tool length
syms g real;                    % Gravity

%% N-E 1 D-H parameter table
DH = [q(1)          0.352         0.070   -pi/2;
      (pi/2)+q(2)   0             -0.360  0;
      q(3)          0             0       pi/2;
      q(4)          0.380         0       -pi/2;
      q(5)          0             0       pi/2;
      q(6)          0.065+Lh      0       0];

[rows,columns] = size(DH);

%% N-E 2 Rotation matrices and their inverses
R = sym(zeros(3,3,rows+1)); % 3D matrix with all rotation matrices
                            % including one representing the rotation until
                            % the application point of an external force

Rinv = R;    % Inverse rotation matrices

for i = 1:rows
    R(:,:,i) = [cos(DH(i,1))    -cos(DH(i,4))*sin(DH(i,1))
sin(DH(i,4))*sin(DH(i,1));
                sin(DH(i,1))    cos(DH(i,4))*cos(DH(i,1))    -
sin(DH(i,4))*cos(DH(i,1));
                0               sin(DH(i,4))                 cos(DH(i,4))];

    Rinv(:,:,i) = inv(R(:,:,i));
end

R(:,:,rows+1) = eye(3);
Rinv(:,:,rows+1) = eye(3);

R = simplify(R);
Rinv = simplify(Rinv);

%% N-E 3 Initial conditions

% Base reference system {S_0}

omega = sym(zeros(3,rows+1));     % Angular velocities (0,1,2,3,4,5,6)
omegad = sym(zeros(3,rows+1));    % Angular accelerations (0,1,...,6)
v = sym(zeros(3,rows+1));         % Linear velocities (0,1,...,6)
vd = sym(zeros(3,rows+1));        % Linear accelerations (0,1,...,6)
vd(3,1) = g;                      % Gravity along axis z
```

```
% Link masses [kg]
m = [34.56013977 15.99814611 16.97933464 3.739855737...
     0.303969 0.054966342]';

% Auxiliary vector z

z = [0 0 1]';

% Coordinates of vector {S_i-1} to {S_i} expressed in {S_i} [m]

p = zeros(3,rows);

for i = 1:rows
    p(:,i) = [DH(i,3); DH(i,2)*sin(DH(i,4)); DH(i,2)*cos(DH(i,4))]';
end

% Coordinates of center of mass i with respect to {S_i} [m]

s = [-0.0419443, 0.0886819, 0.0438186;
     0.161989, 0.00979472, -0.0921779;
     0.00570679, 0.007527, 0.0195804;
     0.00142883, 0.0682882, -0.00122517;
     0.000164849, 0.000619119, -0.00140587;
     0.000224358, -3.08768e-07, -0.0129399]';

% Inertia matrices of link i with respect to its center of mass expressed
% in {S_i} [kg·m^2]

MI = zeros(3,6);    % Moments of Inertia
PI = zeros(3,6);    % Products of Inertia

MI(:,1) = [0.508524, 0.456961, 0.461318]';
PI(:,1) = [0.068918, -0.00146701, 0.0523087]';

MI(:,2) = [0.0952241, 0.327313, 0.276049]';
PI(:,2) = [-0.00105662, -0.0382749, -0.00384372]';

MI(:,3) = [0.182447, 0.199252, 0.0685502]';
PI(:,3) = [0.00640664, -0.00402342, -0.000769641]';

MI(:,4) = [0.0172493, 0.00733781, 0.0153991]';
PI(:,4) = [-0.000436025, -6.90193e-06, -0.000124383]';

MI(:,5) = [0.000145763, 0.000238175, 0.000151397]';
PI(:,5) = [-2.64639e-07, -8.71253e-07, 2.05448e-08]';

MI(:,6) = [1.33687e-05, 1.31196e-05, 1.27092e-05]';
PI(:,6) = [1.31964e-10, -9.71246e-08, 3.18105e-10]';

I = zeros(3,3,6);

for i = 1:rows
    I(:,:,i) = diag(MI(:,i));   % Diagonal is filled

    I(1,2,i) = PI(3,i);
    I(1,3,i) = PI(2,i);
    I(2,3,i) = PI(1,i);
```

```
    I(2,1,i) = PI(3,i);
    I(3,1,i) = PI(2,i);
    I(3,2,i) = PI(1,i);
end

%% N-E 4 Calculation of angular velocities of reference systems (omega_i)

for i = 1:rows
    omega(:,i+1) = Rinv(:,:,i)*(omega(:,i)+z*qd(i,1));
end

%% N-E 5 Calculation of angular accelerations of reference syst. (omega_i')

for i = 1:rows
    omegad(:,i+1) =
Rinv(:,:,i)*(omegad(:,i)+z*qdd(i,1))+cross(omega(:,i),z*qd(i,1));
end

%% N-E 6 Calculation of linear accelerations of reference systems (v_i')

for i = 1:rows
    vd(:,i+1) =
cross(omegad(:,i+1),p(:,i))+cross(omega(:,i+1),cross(omega(:,i+1),p(:,i)))+
Rinv(:,:,i)*vd(:,i);
end

%% N-E 7 Calculation of linear accelerations of the center of mass of
%  link i

a = sym(zeros(3,rows));

for i = 1:rows
    a(:,i) =
cross(omegad(:,i+1),s(:,i))+cross(omega(:,i+1),cross(omega(:,i+1),s(:,i)))+
vd(:,i+1);
end

%% N-E 8 Calculation of force exerted on link i

f = sym(zeros(3,rows+1));   % The force exerted externally is zero

for i = rows:-1:1
    f(:,i) = R(:,:,i+1)*f(:,i+1)+m(i)*a(:,i);
end


%% N-E 9 Calculation of torque exerted on link i

n = sym(zeros(3,rows+1));    % The torque exerted externally is zero

for i = rows:-1:1
    n(:,i) = R(:,:,i+1)*(n(:,i+1)+cross(Rinv(:,:,i+1)*p(:,i),f(:,i+1)))+...
             cross(p(:,i)+s(:,i),m(i)*a(:,i))+...

I(:,:,i)*omegad(:,i+1)+cross(omega(:,i+1),I(:,:,i)*omega(:,i+1));
end

%% N-E 10 Calculation of torques exerted on joint i
```

```
tau = sym(zeros(rows,1));

for i = rows:-1:1
    tau(i,1) = n(:,i)'*Rinv(:,:,i)*z;
end

%% Extraction of M,C,G matrices

vars = qdd';
[M,N] = equationsToMatrix(tau,vars);

N = -N;

vars = g;
[G,C] = equationsToMatrix(N,vars);

C = -C;
```

## Code 13. Computed Torque Controller function for the ABB IRB 140

```
function [err,errd,tau] = fCTC(Kp,Kv,des,cur)
% Computed Torque Controller
% Generates a set of torques from current and desired joint coordinates

n_links = 6;     % Number of links
g = 9.80665;    % Gravity

q_des = zeros(n_links,1);
qd_des = zeros(n_links,1);
qdd_des = zeros(n_links,1);

q_cur = zeros(n_links,1);
qd_cur = zeros(n_links,1);
qdd_cur = zeros(n_links,1);

% Places input information into individual ordered vectors

for i = 1:n_links
    q_des(i) = des(i);
    qd_des(i) = des(i+n_links);
    qdd_des(i) = des(i+n_links*2);

    q_cur(i) = cur(i);
    qd_cur(i) = cur(i+n_links);
    qdd_cur(i) = cur(i+n_links*2);
end

% Calculation of errors

err = zeros(6,1);
errd = err;

for i = 1:n_links
    err(i,1) = q_des(i)-q_cur(i);
    errd(i,1) = qd_des(i)-qd_cur(i);
end

% Substitution and calculation of M, C and G (Obtained through N-E)
M = [...    % Large M declaration
C = [...    % Large C declaration
G = [...    % Large G declaration

% Control law
tau = M*(qdd_des+Kv*errd+Kp*err) + C + G*g;
end
```

## Code 14. Newton-Euler algorithm for the obtention of the 2-link robot dynamic model

```
%% Master's Thesis | ABB IRB 140 | Newton-Euler Algorithm | 2-link robot

clear all
close all
clc

q = sym('q',[2 1],'real');      % 2 joint angles
qd = sym('qd',[2 1],'real');    % 2 joint velocities
qdd = sym('qdd',[2 1],'real'); % 2 joint accelerations
Lh = 0;                         % Tool length
syms L I1 I2 Lc1 Lc2 g real;    % Other parameters

%% N-E 1 D-H parameter table
DH = [q(1)  0   L   0;
      q(2)  0   L   0];

[rows,columns] = size(DH);

%% N-E 2 Rotation matrices and their inverses
R = sym(zeros(3,3,rows+1)); % 3D matrix with all rotation matrices
                            % including one representing the rotation until
                            % the application point of an external force

Rinv = R;   % Inverse rotation matrices

for i = 1:rows
    R(:,:,i) = [cos(DH(i,1))    -cos(DH(i,4))*sin(DH(i,1))
sin(DH(i,4))*sin(DH(i,1));
                sin(DH(i,1))    cos(DH(i,4))*cos(DH(i,1))    -
sin(DH(i,4))*cos(DH(i,1));
                0               sin(DH(i,4))                 cos(DH(i,4))];

    Rinv(:,:,i) = inv(R(:,:,i));
end

R(:,:,rows+1) = eye(3);
Rinv(:,:,rows+1) = eye(3);

R = simplify(R);
Rinv = simplify(Rinv);

%% N-E 3 Initial conditions

% Base reference system {S_0}

omega = sym(zeros(3,rows+1));     % Angular velocities (0,1,2)
omegad = sym(zeros(3,rows+1));    % Angular accelerations (0,1,2)
v = sym(zeros(3,rows+1));         % Linear velocities (0,1,2)
vd = sym(zeros(3,rows+1));        % Linear accelerations (0,1,2)
vd(1,1) = g;                      % Gravity along axis x

% Link masses [kg]

m = sym('m',[rows 1],'real');
```

```
% Auxiliary vector z

z = [0 0 1]';

% Coordinates of vector {S_i-1} to {S_i} expressed in {S_i} [m]

p = sym('p',[3 rows],'real');

for i = 1:rows
    p(:,i) = [DH(i,3); DH(i,2)*sin(DH(i,4)); DH(i,2)*cos(DH(i,4))]';
end

% Coordinates of center of mass i with respect to {S_i} [m]

s = [-L+Lc1      -L+Lc2;
     0            0;
     0            0];

% Inertia matrices of link i with respect to its center of mass expressed
% in {S_i} [kg·m^2]

I(:,:,1) = [0    0    0
            0    0    0
            0    0    I1];

I(:,:,2) = [0    0    0
            0    0    0
            0    0    I2];

%% N-E 4 Calculation of angular velocities of reference systems (omega_i)

for i = 1:rows
    omega(:,i+1) = Rinv(:,:,i)*(omega(:,i)+z*qd(i,1));
end

%% N-E 5 Calculation of angular accelerations of reference syst. (omega_i')

for i = 1:rows
    omegad(:,i+1) =
Rinv(:,:,i)*(omegad(:,i)+z*qdd(i,1))+cross(omega(:,i),z*qd(i,1));
end

%% N-E 6 Calculation of linear accelerations of reference systems (v_i')

for i = 1:rows
    vd(:,i+1) =
cross(omegad(:,i+1),p(:,i))+cross(omega(:,i+1),cross(omega(:,i+1),p(:,i)))+
Rinv(:,:,i)*vd(:,i);
end

%% N-E 7 Calculation of linear accelerations of the center of mass of
%  link i

a = sym(zeros(3,rows));

for i = 1:rows
```

```
    a(:,i) =
cross(omegad(:,i+1),s(:,i))+cross(omega(:,i+1),cross(omega(:,i+1),s(:,i)))+
vd(:,i+1);
end

%% N-E 8 Calculation of force exerted on link i

f = sym(zeros(3,rows+1));    % The force exerted externally is zero

for i = rows:-1:1
    f(:,i) = R(:,:,i+1)*f(:,i+1)+m(i)*a(:,i);
end


%% N-E 9 Calculation of torque exerted on link i

n = sym(zeros(3,rows+1));     % The torque exerted externally is zero

for i = rows:-1:1
    n(:,i) = R(:,:,i+1)*(n(:,i+1)+cross(Rinv(:,:,i+1)*p(:,i),f(:,i+1)))+...
             cross(p(:,i)+s(:,i),m(i)*a(:,i))+...

I(:,:,i)*omegad(:,i+1)+cross(omega(:,i+1),I(:,:,i)*omega(:,i+1));
end

%% N-E 10 Calculation of torques exerted on joint i

tau = sym(zeros(rows,1));

for i = rows:-1:1
    tau(i,1) = n(:,i)'*Rinv(:,:,i)*z;
end

%% Extraction of M,C,G matrices

vars = qdd';
[M,N] = equationsToMatrix(tau,vars);

N = -N;

vars = g;
[G,C] = equationsToMatrix(N,vars);

C = -C;
```

## Code 15. Initialization of the testing CTC control loop containing the 2-link robot model

```
%% Master's Thesis | ABB IRB 140 |
% Newton-Euler Algorithm / 2 Link Robot Initialization

% Initial D-H parameters

DH0 = [0    0    0.45    0;
       0    0    0.45    0];

g = 9.80665;     % Gravity
L = 0.45;        % Length of links
I1 = 1.266;      % Inertia
I2 = 0.093;
Lc1 = 0.091;     % Distance to the center of mass
Lc2 = 0.048;
m1 = 23.902;     % Mass
m2 = 3.88;
n_l = 2;         % Number of links

sim CTC_Test_2Links
```

## Code 16. Computed Torque Controller testing function for a 2-Link robot

```
function [q_err,qd_err,tau1,tau2] = fCTC_2Link(des,cur)

L = 0.45;        % Length of links
I1 = 1.266;      % Inertia
I2 = 0.093;
Lc1 = 0.091;     % Distance to the center of mass
Lc2 = 0.048;
g = -9.80665;       % Gravity
m1 = 23.902;     % Mass
m2 = 3.88;
n_l = 2;         % Number of links

q_des = zeros(n_l,1);
qd_des = zeros(n_l,1);
qdd_des = zeros(n_l,1);

q_cur = zeros(n_l,1);
qd_cur = zeros(n_l,1);
qdd_cur = zeros(n_l,1);

for i = 1:n_l
    q_des(i) = des(i);
    qd_des(i) = des(i+n_l);
    qdd_des(i) = des(i+n_l*2);

    q_cur(i) = cur(i);
    qd_cur(i) = cur(i+n_l);
    qdd_cur(i) = cur(i+n_l*2);
end

q_err = [q_des(1)-q_cur(1);
         q_des(2)-q_cur(2)];
qd_err = [qd_des(1)-qd_cur(1);
          qd_des(2)-qd_cur(2)];

M = [m2*L^2 + 2*m2*cos(q_cur(2))*L*Lc2 + m1*Lc1^2 + m2*Lc2^2 + I1 + I2,
m2*Lc2^2 + L*m2*cos(q_cur(2))*Lc2 + I2;
     I2 + Lc2*m2*(Lc2 + L*cos(q_cur(2))),
m2*Lc2^2 + I2];

C = [-L*Lc2*m2*sin(q_cur(2))*qd_cur(2)^2 -
2*L*Lc2*m2*qd_cur(1)*sin(q_cur(2))*qd_cur(2);
     L*Lc2*m2*qd_cur(1)^2*sin(q_cur(2))];

G = [-Lc2*m2*sin(q_cur(1) + q_cur(2)) - L*m2*sin(q_cur(1)) -
Lc1*m1*sin(q_cur(1));
     -Lc2*m2*sin(q_cur(1) + q_cur(2))];

Kp = diag(900); Kv=diag(60);

torques = M*(qdd_des+Kv*qd_err+Kp*q_err)+C+G*g;

tau1 = torques(1);
tau2 = torques(2);
end
```

## Code 17. Pose comparator

```
function err  = fposecomp(p_cur,p_des)
% Compares the current pose with the desired one, creating an error value
err = p_cur-p_des;
end
```

## Code 18. Trajectory generator function

```
function p_des = fposgen(x,t)

p_home = [0.515 0 0.712 sqrt(2)/2 0 sqrt(2)/2 0]';

circle_small = [0.6     0       0.3 0 0 1 0;
                0.55    0.05    0.3 0 0 1 0;
                0.5     0       0.3 0 0 1 0;
                0.55    -0.05   0.3 0 0 1 0]';

circle_medium = [0.65 0     0.3 0 0 1 0;
                 0.55 0.1   0.3 0 0 1 0;
                 0.45 0     0.3 0 0 1 0;
                 0.55 -0.1  0.3 0 0 1 0]';

circle_large = [0.7     0       0.3 0 0 1 0;
                0.55    0.15    0.3 0 0 1 0;
                0.4     0       0.3 0 0 1 0;
                0.55    -0.15   0.3 0 0 1 0]';

if t <= 2
    p_des = p_home + (circle_small(:,1)-p_home).*x;
else
    if (t > 2) && (t <= 4)
        x_cir = 0.55 + 0.05*cos(x*2*pi); % Parametric circle equations
        y_cir = 0 + 0.05*sin(x*2*pi);

        p_des = [x_cir  y_cir 0.3 0 0 1 0]';
    else
        if (t > 4) && (t <= 6)
            p_des = circle_small(:,1) + (p_home-circle_small(:,1)).*x;
        else
            if (t > 6) && (t <= 8)
                p_des = p_home + (circle_medium(:,1)-p_home).*x;
            else
                if (t > 8) && (t <= 10)
                    x_cir = 0.55 + 0.1*cos(x*2*pi);
                    y_cir = 0 + 0.1*sin(x*2*pi);

                    p_des = [x_cir  y_cir 0.3 0 0 1 0]';
                else
                    if (t > 10) && (t <= 12)
                        p_des = circle_medium(:,1) + (p_home-
circle_medium(:,1)).*x;
                    else
                        if (t > 12) && (t <= 14)
                            p_des = p_home + (circle_large(:,1)-p_home).*x;
                        else
                            if (t > 14) && (t <= 16)
                                x_cir = 0.55 + 0.15*cos(x*2*pi);
                                y_cir = 0 + 0.15*sin(x*2*pi);

                                p_des = [x_cir  y_cir 0.3 0 0 1 0]';
                            else
                                if (t > 16) && (t <= 18)
                                    p_des = circle_large(:,1) + (p_home-
circle_large(:,1)).*x;
                                else
```

```
                                    p_des = zeros(7,1);
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end

end
```

## Code 19. Testing trajectory RAPID code

```
MODULE Module1
    CONST robtarget
Home:=[[515,0,712],[0.707106781,0,0.707106781,0],[0,0,0,0],[9E+09,9E+09,9E+
09,9E+09,9E+09,9E+09]];
    CONST robtarget
CircleSmall4:=[[600,0,300],[0,0,1,0],[0,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+
09,9E+09]];
    CONST robtarget
CircleSmall1:=[[550,50,300],[0,0,1,0],[0,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E
+09,9E+09]];
    CONST robtarget
CircleSmall2:=[[500,0,300],[0,0,1,0],[0,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+
09,9E+09]];
    CONST robtarget CircleSmall3:=[[550,-
50,300],[0,0,1,0],[0,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
    CONST robtarget
CircleMedium4:=[[650,0,300],[0,0,1,0],[0,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E
+09,9E+09]];
    CONST robtarget
CircleMedium1:=[[550,100,300],[0,0,1,0],[0,0,0,0],[9E+09,9E+09,9E+09,9E+09,
9E+09,9E+09]];
    CONST robtarget
CircleMedium2:=[[450,0,300],[0,0,1,0],[0,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E
+09,9E+09]];
    CONST robtarget CircleMedium3:=[[550,-100,300],[0,0,1,0],[-1,0,-
1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
    CONST robtarget
CircleLarge4:=[[700,0,300],[0,0,1,0],[0,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+
09,9E+09]];
    CONST robtarget
CircleLarge1:=[[550,150,300],[0,0,1,0],[0,0,0,0],[9E+09,9E+09,9E+09,9E+09,9
E+09,9E+09]];
    CONST robtarget
CircleLarge2:=[[400,0,300],[0,0,1,0],[0,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+
09,9E+09]];
    CONST robtarget CircleLarge3:=[[550,-150,300],[0,0,1,0],[-1,0,-
1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
    !*********************************************************
    ! Testing Trajectory
    !*********************************************************
    PROC main()
        SmallCircle;
        WaitTime(1);
        MediumCircle;
        WaitTime(1);
        LargeCircle;
    ENDPROC
    PROC SmallCircle()
        MoveJ Home,v300,fine,tool0\WObj:=wobj0;
        MoveJ CircleSmall4,v300,fine,tool0\WObj:=wobj0;
        MoveC CircleSmall1,CircleSmall2,v300,fine,tool0\WObj:=wobj0;
        MoveC CircleSmall3,CircleSmall4,v300,fine,tool0\WObj:=wobj0;
        MoveJ Home,v300,fine,tool0\WObj:=wobj0;
    ENDPROC
    PROC MediumCircle()
        MoveJ Home,v300,fine,tool0\WObj:=wobj0;
        MoveJ CircleMedium4,v300,fine,tool0\WObj:=wobj0;
        MoveC CircleMedium1,CircleMedium2,v300,fine,tool0\WObj:=wobj0;
        MoveC CircleMedium3,CircleMedium4,v300,fine,tool0\WObj:=wobj0;
```

```
        MoveJ Home,v300,fine,tool0\WObj:=wobj0;
    ENDPROC
    PROC LargeCircle()
        MoveJ Home,v300,fine,tool0\WObj:=wobj0;
        MoveJ CircleLarge4,v300,fine,tool0\WObj:=wobj0;
        MoveC CircleLarge1,CircleLarge2,v300,fine,tool0\WObj:=wobj0;
        MoveC CircleLarge3,CircleLarge4,v300,fine,tool0\WObj:=wobj0;
        MoveJ Home,v300,fine,tool0\WObj:=wobj0;
    ENDPROC
ENDMODULE
```

## Code 20. Interpolated data caller function

```
function p_des = fintpol(t)

p_des = zeros(7,1);

s = load('cfits.mat');

p_des(1) = s.x_t(t)*1e-3; % [mm] to [m]
p_des(2) = s.y_t(t)*1e-3;
p_des(3) = s.z_t(t)*1e-3;
p_des(4) = s.a_t(t);
p_des(5) = s.b_t(t);
p_des(6) = s.c_t(t);
p_des(7) = s.d_t(t);

end
```