# Impedance control

**Truls Nilsson**[1] **Jakob Wisth**[2] **Christoffer Gyllenstierna**[3] **Carl Erliden**[4]

Project Advisor: **Marko Guberina**[5]

[1]tr8683ni-s@student.lu.se [2]ja3005wi-s@student.lu.se [3]ch1165gy-s@student.lu.se
[4]ca2633er-s@student.lu.se [5]marko.guberina@control.lth.se

***Abstract:*** This project focuses on improving the performance of a 6-joint robotic arm through the application of impedance control. The goal is to make the robotic arm mimic the behavior of a spring, enhancing its ability to handle errors more effectively. The control system comprises a proportional (P) controller and a filter designed to reduce noise from the force sensor.
The project successfully achieves the desired behavior of the robotic arm, with particular attention given to addressing issues such as joint vibrations through adjustments to specific settings. The practical implementation and careful tuning of the control system demonstrate its effectiveness and adaptability to various situations. This work provides valuable insights for enhancing the precision and resilience of robotic arms in real-world tasks.

## Contents

## 1. Introduction

In today's developing world, robotics is a fast-growing sector with a multitude of applications. One of these applications is the automation of various tasks in industries. These tasks require that a robot can execute precise and exact movements; however, the models of the robots are not always precise, and errors do occur. One solution to make a robot more resilient to these errors is to introduce impedance control. Impedance control is a way to enhance safety for both the completion of the task and the robot itself by adding one or multiple force sensors to give the robot the ability to feel its environment.
In this project, impedance control has been written and implemented for a robotic arm to make it behave like a spring. To demonstrate the functions of impedance control, this project will implement it for both a single stationary point and a moving task.

## 2. Software tools

The control of the robot involved the use of the communication interface *ur_rtde* and the robot dynamics library *Pinocchio*, in conjunction with the provided robot model. The operating system used for this project was Ubuntu 22 LTS and *Python* served as the coding environment, with the *numpy* library employed for mathematical calculations.

### 2.1 RTDE Interface

The *ur_rtde* interface provides the user with the possibility to control the robot using predefined commands from Universal Robots. These commands can, for example, control the angle position, angular velocity, and angular acceleration of the joint. The receive interface can, for example, provide the user with the current position and velocities of all joint angles, along with the force applied to the end-effector. The commands used for this project are listed in the appendix.
In order to test the robot without damaging it through errors in code or by commanding joints to move to impossible positions, the RTDE interface has the possibility to connect to a simulated environment and can be seen in figure (1) .

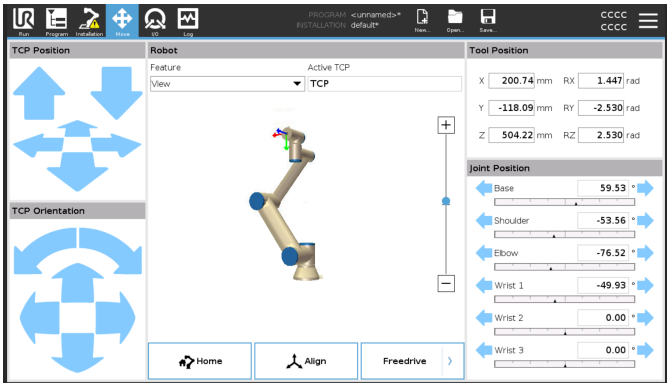


**Figure 1.** Simulated environment used for testing.

This is where most of the testing of the code is done. Since the final goal is to make the robotic arm to react to external forces it is easier to test the force sensor on the real robotic arm.

### 2.2 Pinocchio

*Pinocchio* is an open source library that is used for robot dynamics and calculations. It will be used for handling the robot parameters, calculating the forward kinematics, converting joint space to Cartesian space and calculating the Jacobian matrix of the joint angles.

### 2.3 clik

To be able to get the robot to move to a desired position the function $moveJ()$ from the *ur_rtde* library can be used but the input position is in joint space which is not very intuitive to a human. On otherhand the command is very useful for setting a home position of the robot arm. Instead of using the $moveJ()$ command, the command $speedJ()$ combined with an algorithm called *clik* can be used. The name *clik* is an abbreviation for "closed loop inverse kinematic". It makes the robotic arm to move to a specific position in Cartesian space using something called inverse kinematics which will be later explained in chapter 3. This code was provided by the school and can also be found on Pinocchios website.

## 3. Robot dynamics modeling

The Universal Robots UR5e, seen in Figure (2), is a robotic arm designed for a variety of applications. It consists of 6 revolving joints, seen as the blue area of the robot. These joints can rotate freely around their axes. The outermost part of the arm is called the end-effector. By combining different angles of all joints, the position of the end-effector can be controlled.

**Figure 2.** UR5e robotic arm.

Robot dynamics is a crucial aspect when working with robotic systems, as it involves understanding the motion, forces, and torques associated with the robot's movement. Dynamics plays a key role in tasks such as trajectory planning, control, and, in this case, impedance control.

### 3.1 Forward kinematics

Kinematics is the relationship between a robot's structure and its movement in Cartesian space, considering the robot's configuration and joint lengths. The configuration defines the state of the controlled robot parts. In the case of a robot arm, this involves the state of its motors, known as revolving joints. By tracking how much a motor turns from its starting point or with position sensors mounted on the motor shaft, it's possible to keep track of the angle of the joint. For a robot equipped with six joints, the angles are represented using a six-dimensional

vector q, where each entry indicates the rotation of a joint in radians, e.g., $q = [\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6]^\intercal$, and can also be seen in marked on the robot arm in Figure (3).

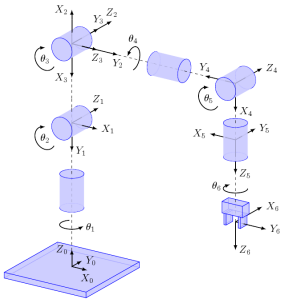

**Figure 3.** 6 joint robotic arm with joint angles and coordinate axles marked.

The goal of the forward kinematics algorithm is to calculate the position of the end-effector defined as a matrix $T(q)_{O,ee}$, which is the transformation matrix from the origin (O) to the end-effector (ee). The transformation matrix includes both rotation and translation matrices, as seen in Equation 1.

$$\begin{bmatrix} X_{ee} \\ Y_{ee} \\ Z_{ee} \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_O \\ Y_O \\ Z_O \\ 1 \end{bmatrix} \tag{1}$$

The rotation matrix, noted as $r_{nm}$, describes the roll, pitch, and yaw angle relations between the two coordinate systems $O$ and $ee$. The translation vector, noted as $t_x, t_y, t_z$, describes the position of the origin of the systems in relation to each other. The transformation matrix is that the transformation matrix from the origin to the end-effector, $T_{O,ee}$, is equal to all transformation matrices of the joints multiplied together, as seen in Equation 2.

$$T_{O,ee} = T_{O,1} \cdot T_{1,2} \cdot T_{2,3} \cdot T_{3,4} \cdot T_{4,5} \cdot T_{5,ee} \tag{2}$$

As the angles of all six joints are known due to the position sensors, and the lengths of the links between the joints are constant and defined, the transformation matrix can always be calculated. These calculations are performed by the Pinocchio library, as mentioned in an earlier section 2. [1][3]

### 3.2 Inverse Kinematics

From the forward kinematics algorithm, the position of the end-effector can be calculated from the current robot configuration. However, most of the time, this is not what the user wants. Instead, the user wants to instruct the robot on where the end-effector should be, and the robot should move to that position. This process is known as inverse kinematics, which is a much more complicated algorithm. The complexity arises from the multiple or even infinite number of solutions to a given position. For a 2DOF robotic arm, there are two solutions to the position of the end-effector, as illustrated in Figure 4.

With an increasing number of joints and degrees of freedom, the number of solutions increases, and due to the rotational matrix, there is an almost infinite number of solutions

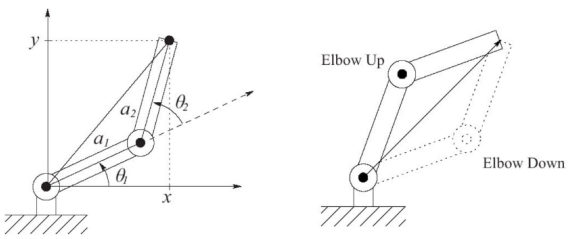


**Figure 4.** Inverse kinematics of a 2DOF robotic arm.

to a point in 3D space. A solution to this complexity is the algorithm called $clik$, as mentioned earlier. Consider a goal of moving the end-effector from an initial point $p$ to the point $p^*$, where both points are expressed in the world frame. With the robot configuration $q$, the task is formulated as the following equation.

$$e(q) = p^* - p(q) \tag{3}$$

Where the goal is to bring the error, $e(q)$, to zero. The point $p(q)$ can be calculated using the previously mentioned forward kinematics, and from that the Jacobian matrix can be computed:

$$J(q) = \frac{\partial p}{\partial q}(q) \tag{4}$$

which maps the joint angle velocities $\dot{q}$ to the end-effector velocity $\dot{p}$ with the rewritten equation.

$$J(q)\dot{q} = \dot{p}(q) \tag{5}$$

The Jacobian matrix will be explained in more detail in the next section. By applying a velocity $\dot{q}$ over a short time interval $\Delta t$ the new error then becomes.

$$e'(q) = e(q) - \dot{p}(q) \cdot \Delta t \tag{6}$$

With this new equation, the goal is to set $e'$ to zero, which would imply that $e(q) = \dot{p}(q) \cdot \Delta t = \frac{p^* - p(q)}{\Delta t} \cdot \Delta t$, leading to a definition of the velocity error.

$$v(q, \Delta t) = \frac{e(q)}{\Delta t} = \frac{p^* - p(q)}{\Delta t} \tag{7}$$

According to these equations, the goal is to set the joint velocities $\dot{q}$ such that:

$$J(q)\dot{q} = \dot{p}(q) = v(q, \Delta t) \tag{8}$$

Since $J(q)$ and $\dot{p}(q)$ are known from previous equations, multiplying both sides by $J^{-1}$ gives the values of the joint velocities, as shown in the following equation.

$$\dot{q} = J^{-1} \cdot \dot{p}(q) \tag{9}$$

Unfortunatly, this is not always possible due to that the Jacobian matrx is not always invertable, but it can be solved using the pseudoinverse, which is stated as the following equation.

$$(J^T \cdot J)q = J^T \cdot \dot{p}(q) \tag{10}$$

To avoid problems with singularities, such as joints colliding, the damped pseudoinverse can be used instead, written as the following equation:

$$\dot{q} = -J^T(JJ^T + \lambda I)^{-1}\dot{p}(q) \tag{11}$$

where $\lambda$ is the damping factor which for this project was chosen through testing. [1] [3]

### 3.3 Dynamic Equations

The dynamics of a robot are often described by the equations of motion, which relate joint positions, velocities, accelerations, and external forces/torques.

In this case, the robot will be exposed to external forces and the response will be controlled by the impedance control. The robots dynamic equations have the general form of:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = u \tag{12}$$

$M(q)$ is the symmetric positive definite mass inertia matrix of the system, $C(q,\dot{q})$ is the matrix of Corolis and centrifugal terms, $g(q)$ is the vector of gravity terms and $u$ is the input vector. From this, the inverse dynamics can be expressed as:

$$\ddot{q} = M^{-1}(q)(u - C(q,\dot{q})\dot{q} - g(q)) \tag{13}$$

**The $M(q)$ matrix** can be calculated as

$$M(q) = \left[\sum_{i=0}^{n}(m_i J_{v_i}^T J_{v_i} + J_{\omega_i}^T R_i I_i R_i^T J_{\omega_i})\right] \tag{14}$$

where $J_{v_i}$ and $J_{\omega_i}$ are the linear and angular part of the Jacobian matrix $J_i$, respectively.

The Jacobian matrix is a mathematical tool used in many different applications and robotics is one of them. In robotics it is used to relate joint velocities to its end-effector velocities. It provides a mapping between the rate of change in joint angles to linear and angular velocities of the end-effector. The Jacobian matrix is often represented as [2]:

$$J = \begin{bmatrix} J_{\text{linear}} \\ J_{\text{angular}} \end{bmatrix} \tag{15}$$

1. Linear part $J_{v_i}$:
   - The linear part of the Jacobian relates the linear velocities of the end-effector to the joint velocity. Or in other words, how a small change in the joint velocity affect the linear position of the end-effector.
   - The linear part of the Jacobian is often denoted as $J_{v_i} = \frac{\delta \text{position}}{\delta \text{joint angle}}$
2. Angular part $J_{\omega_i}$:
   - The angular part of the Jacobian relates the angular velocities of the end-effector to the joint velocities. As the name suggests, it describes how a small change in joint position affect the rotational orentiation off the end-effector.
   - The angular part of the Jacobian is often denoted as: $J_{\omega_i} = \frac{\partial \text{orentation}}{\partial \text{joint angle}}$

The full Jacobian matrix combines both the linear and angular parts to represent the overall relationship between joint velocities and end-effector velocities. It enables the robot controller to compute the required joint velocities to achieve a desired end-effector motion or force.

**The $C(q,\dot{q})$ matrix** is a function of joint positions (q) and joint velocities ($\dot{q}$). It can be derived from the inertia matrix M(q) using Christoffel symbols.

$$C(q,\dot{q}) = \sum_{k=1}^{6} c_{ijk}\dot{q}_j\dot{q}_k \tag{16}$$

where $c_{ijk}$ are the Christoffel symbols, defined as:

$$c_{ijk} = \frac{1}{2}\left(\frac{\delta M_{ij}}{\delta q_k} + \frac{\delta M_{ik}}{\delta q_j} - \frac{\delta M_{jk}}{\delta q_i}\right) \tag{17}$$

**The elements of the gravity vector $g(q)$**

$$g_i(q) = \frac{\delta \mathcal{P}}{\delta q_i} \tag{18}$$

[2]. If the reader wants to know more about things like this, read [3].

## 4. Control

In the preceding section, the general form of motion (12) and the inverse dynamics (13) were revisited. The intended control mechanism for this robot involves the use of speed rather than torque, departing from the more conventional approach. The chosen control principle for the robot arm is based on a straightforward P (proportional) controller, complemented by a filter to mitigate the noise originating from the force sensor. This configuration has proven effective for the fundamental implementation of impedance control, particularly for a stationary point.

### 4.1 Implementation

The implementation of the P controller for impedance in the robot arm involved utilizing the Python programming language. Python was preferred over C++ for two primary reasons. Firstly, Python offers easier execution compared to C++, which is advantageous for making incremental adjustments, such as optimizing control parameters. Secondly, the team possessed more expertise in Python than in C++, making it the pragmatic choice to minimize project startup time.

The final code's control loop, along with calculations for forces from the force sensor, can be seen on the GitLab page provided in the appendix. The control loop includes an offset to compensate for drifting forces, calculated at the program's start using the average of 2000 samples. Additionally, a signal noise filter was implemented using a moving average filter with a weight factor $\beta$ of 0.7. The impedance matrix $Z$ is multiplied with the force matrix (wrench) to introduce the impedance forces. The impedance matrix scales the robotic system's stiffness along the x, y, and z axes, as well as rotational

movements around these axes. The introduction of a zero value in any given axis or rotational dimension serves to nullify the force signal associated with that particular parameter, thereby rendering the robot motionless in the affected axis.

$$Z = \begin{bmatrix} x & 0 & 0 & 0 & 0 & 0 \\ 0 & y & 0 & 0 & 0 & 0 \\ 0 & 0 & z & 0 & 0 & 0 \\ 0 & 0 & 0 & rx & 0 & 0 \\ 0 & 0 & 0 & 0 & ry & 0 \\ 0 & 0 & 0 & 0 & 0 & rz \end{bmatrix} \tag{19}$$

The wrench matrix is then converted from base coordinates to end-effector coordinates. To get the desired forces for impedance control, Z and wrench are multiplied together as:

$$\text{wrench} = \begin{bmatrix} x & 0 & 0 & 0 & 0 & 0 \\ 0 & y & 0 & 0 & 0 & 0 \\ 0 & 0 & z & 0 & 0 & 0 \\ 0 & 0 & 0 & rx & 0 & 0 \\ 0 & 0 & 0 & 0 & ry & 0 \\ 0 & 0 & 0 & 0 & 0 & rz \end{bmatrix} \cdot \text{wrench} \tag{20}$$

To implement impedance control in the provided speed-regulated robot arm, modifications were made to the general dynamics equation. This involved retaining only the terms accessible through either the external force sensor at the end-effector or the robot interface library. The impedance equation was then structured to resemble a damped spring system:

$$\text{Force} = -K \cdot (x_d - x) - \beta \cdot (\dot{x}_d - \dot{x}) \tag{21}$$

Where K and $\beta$ are constants, while x, $x_d$, $\dot{x_d}$, and $\dot{x}$ represent the position, desired position, desired velocity, and velocity, respectively. Transforming the general dynamics equation (12) into this form yields the following equation:

$$M \cdot (\ddot{x}_d - \ddot{x}) + K \cdot (\dot{x}_d - \dot{x}) + \beta \cdot (x_d - x) = F_a \tag{22}$$

Given that the robot arm is speed-controlled, the equation required further adjustment. Integration and modifications were applied to achieve a form implementable on the robot arm. The final implemented equation takes the form:

$$v = K_p \cdot (q_{\text{init}} - q) + \alpha \cdot \tau - K_v \cdot \dot{q} \tag{23}$$

Here, $K_p$ represents the proportional gain, $q_{\text{init}}$ is the initial position for the robot arm (expressed in angles for the six different joints in radians), $q$ is the current joint position, $\alpha$ is a constant determining the influence of forces from the force sensor on the system, $\tau$ represents the forces from the force sensor multiplied by the transpose of the Jacobian matrix, $K_v$ is a damping constant, and $\dot{q}$ is the current joint velocity in radians per second.

The last phase of the project involved integrating impedance control with circular motion. Initially, the circular path had to be calculated. As the impedance control algorithm operates in joint space, the path had to be represented in joint space as well. Attempts to calculate the path in Cartesian space and then convert it to joint space resulted in significant errors. Instead, the clik algorithm was modified and used. The modified clik algorithm iterates through a circular path, sampling and saving different joint angles to a list.

When combining the circular path with impedance control, one can iterate over that list for the desired position variable. To ensure that the robot continues to use impedance control around the current point in the circular path and does not move on to the next point prematurely, an error variable was introduced. With this, the robot cannot move on to the next point on the list until it's close enough to the current one.

## 5. Results

The group conducted successful tests on the impedance control system implemented on the physical robot, and overall, it functions as intended. However, a notable concern emerged in the form of vibrations within the joints. To address this issue, the group tried reducing the maximum acceleration setting. While this adjustment led to slower arm movement, it proved effective in substantially minimizing the undesired shaking, approaching near elimination.

When combining impedance control with a circular motion, some problems occurred. In the initial attempt, the path was calculated simultaneously as the impedance control was running. This resulted in a very slow program and significant errors, causing the arm to drift away. By instead implementing the path calculation in clik and then iterating through the calculated points without impedance control, it was possible to sample the positions of the joint angles while it was running. This was tested in the simulation environment and plotted in graphs, visible in Figure (5) and Figure (6).

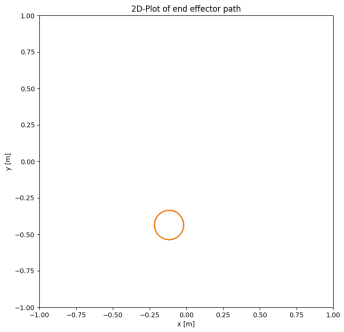


**Figure 5.** 2D-plot of the simulated circular path of the end-effector.

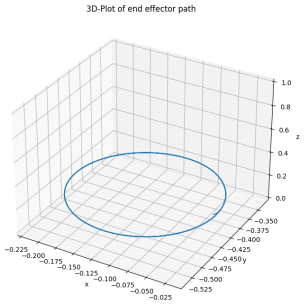


**Figure 6.** 3D-plot of the simulated circular path of the end-effector.

These sampled points were used as the path for the impedance control, as shown in Figure (7).

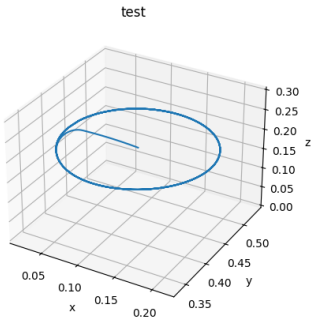


**Figure 7.** 3D-plot of the simulated circular path and impedance control of the end-effector with a radius of 0.1m.

Since it's impossible to push or pull the end-effector in the simulated environment, the path forms a perfect circle, similar to the clik algorithm. The movement from the center of the circle to the edge represents the initial step to reach the desired radius. This posed an issue in the clik algorithm, causing it to sample the path from the center to the edge. When using this path in the impedance control algorithm, which loops through the path multiple times, the arm returned to the center point every revolution. This issue was addressed by introducing an error constraint to ensure a full rotation before starting to sample the angles. When the clik algorithm was applied to the real robot, the results were comparable to the simulated environment, as shown in Figure (8) and Figure (9).

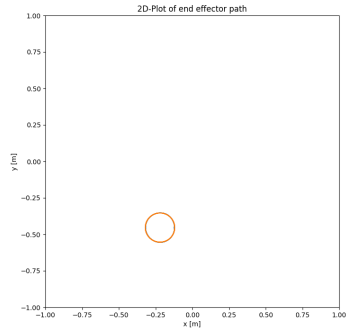


**Figure 8.** 2D-plot of the circular path of the real robots end-effector.

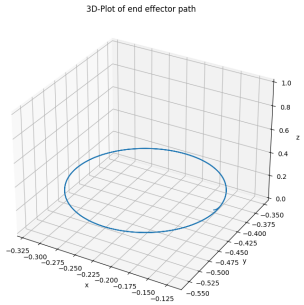


**Figure 9.** 3D-plot of the circular path of the real robots end-effector.

Despite the sampled path being a perfect circle, the combination with the impedance control on the real robot encountered some errors. The most notable issue was that the circular path became slightly elliptical when applied to the real robot. Another challenge was that a relatively large error margin for inverse kinematics was required for the robot to move to the next point in the circle. The path can be seen in Figure (10).

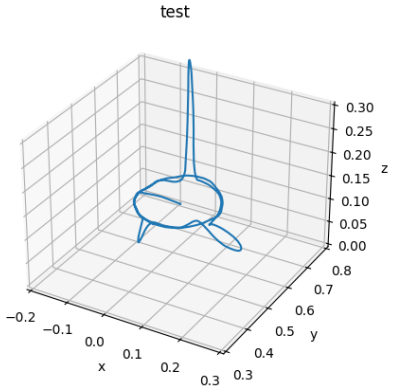


**Figure 10.** 3D-plot of the real circular path and impedance control while pushing and pulling the end-effector.

As seen in Figure (10), the end-effector finds its way back to the elliptical path after disturbance from external forces.

## 6. Discussion

This project significantly enhanced our team's proficiency in utilizing Linux for diverse applications. We delved into mathematical concepts and dynamics previously unfamiliar to many members, with a substantial portion of our efforts devoted to mastering Linux commands for tasks typically performed on a conventional desktop, executed within a Linux terminal. Simultaneously, we focused on understanding the theory of impedance control.

While our employed controller may not be deemed advanced, its integration posed initial challenges that demanded collaborative problem-solving to piece together a functional unit. One persistent challenge that was not fully resolved relates to undesirable shaking and vibrations within the joints. This problem resulted from the fast acceleration of joint angles contributing to an oscillation around the desired position. Although reducing acceleration has provided a partial remedy, a complete elimination of this issue was never accomplished.

The more advanced task, where the robot end-effector should move in a circular path while getting disturbed by an external force, was also accomplished. However, this task faced some problems that we didn't succeed in solving. One of them was the morphing of the circle into an ellipse, a behavior not observed in the simulated environment. This could be related to noise from the force sensor, but since it consistently follows the same elliptical path through multiple revolutions, this may not be the case. Using a better filter than a moving average could be explored to test this hypothesis.

Another problem with the combination of impedance control and a moving end effector was that while iterating through the path points, it sometimes got stuck in a single point and never reached the error margin. This caused it to stay still around this point until it received a small push in the right direction. Attempts to eliminate this by adding an integrator to the control system made the overall system more unreliable and did not solve the issue. This problem may also be related to noise from the force sensor. Increasing the error margin almost eliminated this issue but decreased the accuracy of the end-effector position.

## References


[1] *Inverse kinematics*. URL: https://scaron.info/robotics/inverse-kinematics.html (visited on 2024-01-03).

[2] P. Kebria, S. Al-Wais, H. Abdi, and S. Nahavandi. "Kinematic and dynamic modelling of ur5 manipulator". In: 2016, pp. 004229–004234. DOI: 10.1109/SMC.2016.7844896.

[3] *Robotics explained*. URL: https://robotics-explained.com (visited on 2024-01-03).

## 7. Appendix

Control commands for RTDE interface:

- moveJ(q[]):
  Moves the robot to the joints angle position q
- speedJ(velocity[], acceleration, timeout):
  Sets a desired angular velocity to each joint with a constant acceleration for all joints.
- freedriveMode()
  Sets the robot in free drive mode which let's the user to move the robot freely by hand

Receive commands:

- getActualQ():
  Returns the angle position of the joints as a vector.
- getActualQd():
  Returns the angular velocity of the joints as a vector.
- getActualTCPForce():
  Returns the forces applied to the end-effector as a vector.

Gitlab repositories:

- Impedance Control:
  https://gitlab.control.lth.se/regler/frtn70/2023ht/group-a/-/blob/master/examples/impedence_control.py?ref_type=heads
- Circular motion Impedance Control:
  https://gitlab.control.lth.se/regler/frtn70/2023ht/group-a/-/blob/master/examples/impedance_control_circular_motion.py?ref_type=heads