DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Report for Practical Course

Control of Modular Robots

Authors:	Sophie Sepp, Benedict Feldotto, Abdallah Emad Attawia, Daniel Heinze
Supervisor:	Andrea Giusti M.Sc.
Submission Date:	02.07.2015

Abstract

In this course our team implements the kinematic and dynamic model of Schunk's reconfigurable modular robot manipulator LWA 4P. The forward and inverse kinematics are implemented and trajectories in the task space are designed. For the designed trajectories a kinematic control algorithm is implemented and tested. For the dynamical model we use Matlab and Simulink to implement the single modules. For that we first obtain the Equations of Motion using the Newton-Euler-formulation. We characterize the modules for an automatic modelling method by providing estimates for masses and inertia tensors using measurements and CAD data, so that the EoM can be automatically achieved. We then provide the preliminary Simulink model for the robotic arm. Secondly, we identify the dynamic model of the actuators using available data. We implement a simulator of the robot by enhancing the Simulink model including saturation effects and possible delays and validate the model using measurements. The third step is to design and tune the decentralized controllers while considering limitations due to the sampling time and test them on the robot. We finally test the controllers with a trajectory planner modelling the kinematic behaviour, which is developed in the kinematic model.

Contents

1 Introduction

As robots are getting more and more complicated, modular robots play an important role in robotics. Modular robots consist of independent cubes, which can connect themselves in different ways in order to fulfill a task. Each of the cubes has its own power supply and intelligence and can have specific tools for solving a problem under special conditions. After a computer has been fed a problem, it analyzes it and connects to the cubes in such a way that it can solve the problem. The cubes have the same size in modular robots in opposite to fractal robots, which can have cubes of different sizes. The cubes can be practically identical built, with similar software implementations, so that the input data and combinations of moduls determines how the task is executed. In this course our team implements the dynamic model of Schunk‘s reconfigurable modular robot manipulator LWA 4P. We use Matlab and Simulink to implement the single modules. For that we first obtain the Equations of Motion using the Newton-Euler-formulation. We characterize the modules for an automatic modelling method by providing estimates for masses and inertia tensors using measurements and CAD data, so that the EoM can be automatically achieved. We then provide the preliminary Simulink model fort he robotic arm. Secondly, we identify the dynamic modell oft he actuators. With Black-Box identification and with White-Box identification. In the next step, we implement a simulator of the robot by enhancing the simulink model including saturation effects and possible delays and validate the model using measurements. The fourth step is to design and tune the decentralized controllers while considering limitations due to the sampling time and test them on the robot. We finally test the controllers with a trajectory planner modelling the kinematic behaviour, which has been developed by our second team.

2 Kinematic Modelling and Kinematic Control

Responsible: *Abdallah Attawia and Daniel Heinze*
Kinematics is the study of motion of a body or a system of bodies without taking the forces causing this motion into consideration. In this chapter the forward and inverse kinematics as well as the kinematic control of the Schunk Powerball Light Weight Arm LWA 4P are studied using Matlab/Simulink. This manipulator is composed of three powerballs and two links connecting them. Each ball contains two revolute joints, which have joint limits from $[-170\,°, 170\,°]$, and thus, has two degrees of freedom (2.1). In this study there is no end-effector attached to the last coordinate frame of the manipulator. Therefore, the last coordinate frame will have a distance of zero from the one before.

2.1 Forward Kinematics

Author: *Abdallah Attawia* [SK08] - [SSVO10] - [Sch15] - [Cur14] Obtaining the pose of

the end-effector of the robotic arm in the task space relative to the base is called the forward kinematics. By computing the transformation between the coordinate frame of the end-effector (also called the tool-frame) and the base frame, the forward kinematics problem is solved. Deriving the forward kinematics can be done using more than one convention. However, in this study, the Denavit-Hartenberg convention is used.

Starting from the bottom up, the following sections explain how to acquire the pose of the end-effector given the joint angles.

2.1.1 Denavit-Hartenberg Convention

The Denavit-Hartenberg Convention is a straightforward systematic way to obtain the parameters of each joint and to define the relative position and orientation of two consecutive links. Before evaluating the DH parameters, the coordinate frames are assigned to each joint according to the following criterion:

1. Axis z_i is chosen along the axis of joint $i+1$, i.e. z_i is the axis of rotation of joint $i+1$

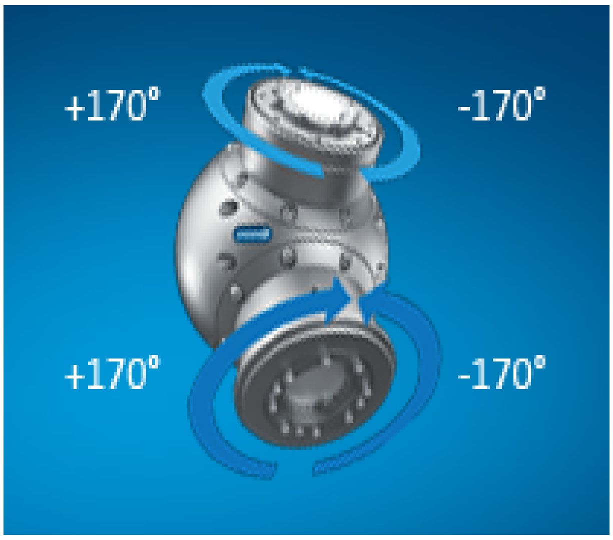


Figure 2.1: Schunk LWA 4P Motion Limits

2. Axis x_i is chosen along the common normal of z_{i-1} and z_i, its in the direction from joint i to joint $i+1$

3. Axis y_i is chosen as to complete the right hand rule

Figure 2.2 shows the assigned coordinate frames to the Schunk LWA 4P manipulator. The distance from the base to the first powerball is $0.205m$. The lengths of the first and second links are $0.350m$ and $0.304m$,respectively.

The DH parameters are a, α, d and θ. $'a'$ is the distance between the joints along the x-axes and α symbolizes the joint rotation about the x-axes. In contrast d and θ define the distance along and rotation about the z-axes, respectively. Out of the four DH parameters, two, a and α, are constant and depend only on the geometry of the manipulator and the assigned coordinate frames. One of the other two parameters is a variable depending on the type of the joint.

- if joint i is revolute, θ_i is a variable

- if joint i is prismatic, d_i is a variable

The six joints of the Schunk LWA 4P manipulator are revolute, which means that θ is the only input variable for each joint. Table 2.1 shows the DH parameters of the Powerball, which are not unique since they depend on the coordinate frame assignments.

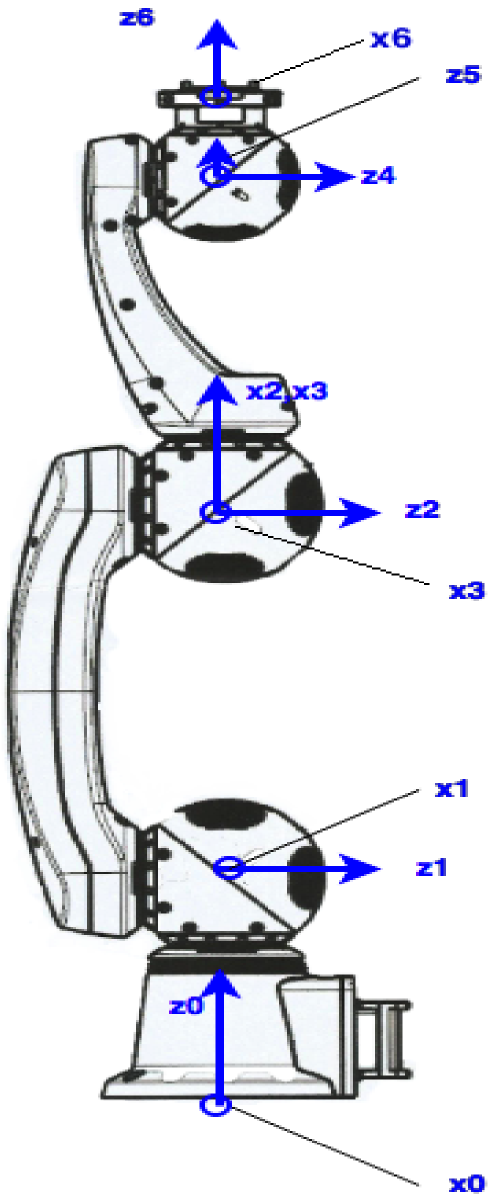


Figure 2.2: Schunk LWA 4P Assigned Coordinate Frames

i	a_i	α_i	d_i	θ_i
1	0	$-\pi/2$	0.205	θ_1
2	0.350	0	0	θ_2
3	0	$\pi/2$	0	θ_3
4	0	$-\pi/2$	0.305	θ_4
5	0	$\pi/2$	0	θ_5
6	0	0	0	θ_6

Table 2.1: DH Parameters of the Schunk LWA 4P Manipulator

Acquiring the Denavit-Hartenberg parameters is the the first step to solve the forward kinematics problem. It allows to find the homogeneous transformation matrix, which is explained in the following section. It is taken into account that the joint angle limit for all joints is between $[-170\,^\circ, 170\,^\circ]$.

2.1.2 Homogeneous Transformation Matrix

The homogeneous transformation matrix is a matrix that represents the spatial displacement of the tool-frame relative to the base frame. It is derived by multiplying the transformation matrices of each two consecutive links. 2.1 shows the individual coordinate transformations from joint i to $i-1$.

$$A_i^{i-1}(q) = \begin{bmatrix} cos(q_i) & -sin(q)_i cos\alpha_i & sin(q_i) sin\alpha_i & a_i cos(q_i) \\ sin(q_i) & cos(q_i) cos\alpha_i & -cos(q_i) sin\alpha_i & a_i sin(q_i) \\ 0 & sin\alpha_i & cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

The DH parameters are substituted into the A-matrix to obtain the transformation between each two consecutive links.

$$A_1^0 = \begin{bmatrix} cos(q_1) & 0 & -sin(q_1) & 0 \\ sin(q_1) & 0 & cos(q_1) & 0 \\ 0 & -1 & 0 & 205.3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

2.2 shows an example of an individual coordinate transformation matrix. A_1^0 is the transformation matrix from frame 1 to frame 0. The individual transformation matrix between each two consecutive coordinate frames are obtained the same way. As shown in 2.3 these individual coordinate transformation matrices are expressed in one homogeneous transformation matrix T_i^0 that relates the end-effector to the base.

$$T_i^0 = A_1^0(q_1) * A_2^1(q_2)......A_i^{i-1}(q_i) \tag{2.3}$$

2.1.3 Position and Orientation Representation

The minimum number of coordinates to represent a pose in the space is six; three coordinates to describe the position and the other three for the orientation. Let $k(q)$ be the pose of the end-effector as a function of the joint variables $[q_1, q_2, ...q_n]$. Let P_e be the position of the end-effector and ϕ_e its orientation. 2.4 shows how the end-effector's pose is represented.

$$k(q) = \begin{bmatrix} P_e \\ \phi_e \end{bmatrix} \tag{2.4}$$

In order to reach the aforementioned representation, the position is extracted and the orientation is computed from the homogeneous transformation matrix 2.5.

$$T_e^0 = \begin{bmatrix} r_{11} & r_{12} & r_{13} & {}^0p_e^x \\ r_{21} & r_{22} & r_{23} & {}^0p_e^y \\ r_{31} & r_{32} & r_{33} & {}^0p_e^z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

First the $3x1$ position vector $'P_e'$ (2.6) is formed by taking out the first three elements of the last column from the homogeneous transformation matrix in 2.5.

$$P_e = \begin{bmatrix} 0p_e^x \\ 0p_e^y \\ 0p_e^z \end{bmatrix} \tag{2.6}$$

As a simplified example for the position vector, looking at the position vector in 2.2 it shows the distance between the base frame 0 and frame 1, which is a translation along the $z0$ with a distance of $0.205m$ (2.7).

$$P_e = \begin{bmatrix} 0 \\ 0 \\ 0.205 \end{bmatrix} \tag{2.7}$$

Then the $3x1$ orientation vector $'\phi_e'$ is formulated by calculating a set of Euler angles 2.8 from the $3x3$ rotation matrix from 2.5. This is also called the minimal representation of the orientation of a body in space. There are two possible sets of Euler angles,

$\theta 1$	$\theta 2$	$\theta 3$	$\theta 4$	$\theta 5$	$\theta 6$
$\frac{-\pi}{4}$	$\frac{\pi}{2}$	$\frac{-\pi}{4}$	$\frac{\pi}{8}$	$\frac{\pi}{6}$	$\frac{\pi}{2}$

Table 2.2: Random Joint Angle Assignments

namely, ZYX (Rollφ-Pitchθ-Yawψ) and ZYZ angles. In this study the Roll-Pich-Yaw method is used. The θ is not to be confused with the joint angle's θ.

$$\phi = \begin{bmatrix} \varphi \\ \theta \\ \psi \end{bmatrix} \tag{2.8}$$

where,

$$\varphi = atan2(r_{21}, r_{11}) \tag{2.9}$$

$$\theta = atan2(-r_{31}, \sqrt{r_{32}^2 + r_{33}^2}) \tag{2.10}$$

$$\psi = atan2(r_{32}, r_{33}) \tag{2.11}$$

2.1.4 Forward Kinematics Test

To test the forward kinematics solution, two configurations are evaluated; the zero configuration, where all the joint angles are set to zero, and another random configuration.

Figure 2.3 shows the Powerball in the zero configuration. The black part represents the base and the blue and the red parts represent link 1 and link 2, respectively.

The second configuration is based on random joint angles shown in table 2.2. These joint angles are substituted into the homogeneous transformation matrix (2.12) and then the end-effector's pose (2.13) is obtained. Moreover, figure 2.4 shows the manipulator in this configuration and the end-effector in this pose.

$$T_e^0 = \begin{bmatrix} -0.191 & -0.15 & 0 & 0.4 \\ 0.191 & 0.15 & 0 & -0.4 \\ 0.271 & 0.919 & 0 & 0.421 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.12}$$

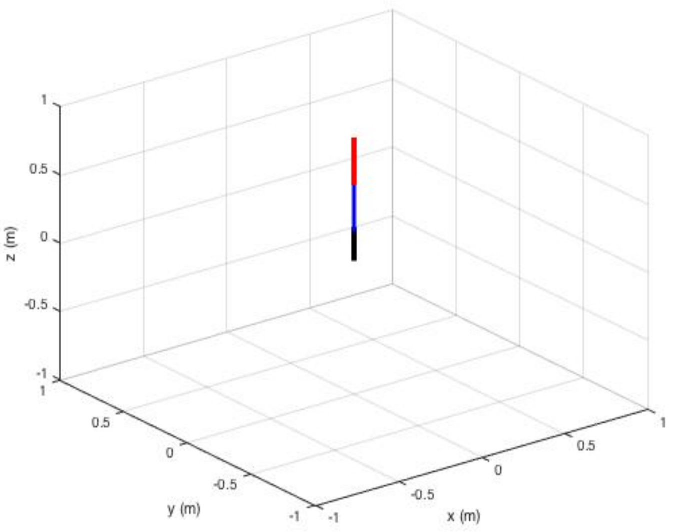


Figure 2.3: Powerball Zero Configuration

$$P_e = \begin{bmatrix} 0.4 \\ -0.4 \\ 0.421 \\ 2.36 \\ -0.286 \\ 1.57 \end{bmatrix} \tag{2.13}$$

2.2 Inverse Kinematics

Author: *Abdallah Attawia* [SK08] - [SSVO10] - [Cra05] The inverse kinematics solution

is required to find out the proper joint angles needed to reach a certain position and orientation of the end-effector. To reach this end, several methods can be used such as Newton-Raphson iterative method, Gradient Descent method, Cyclic Coordinate method, ..etc. However, in this study, the Gradient Descent Method is utilized. Several steps need to be executed in order to solve the inverse kinematics problem, i.e. obtaining the Jacobi matrix (known as the Jacobian), setting random initial joint angles and iterating until the right joint angles are found. To test the inverse kinematics algorithm joint-space trajectories for point to point motion in the task space are designed.

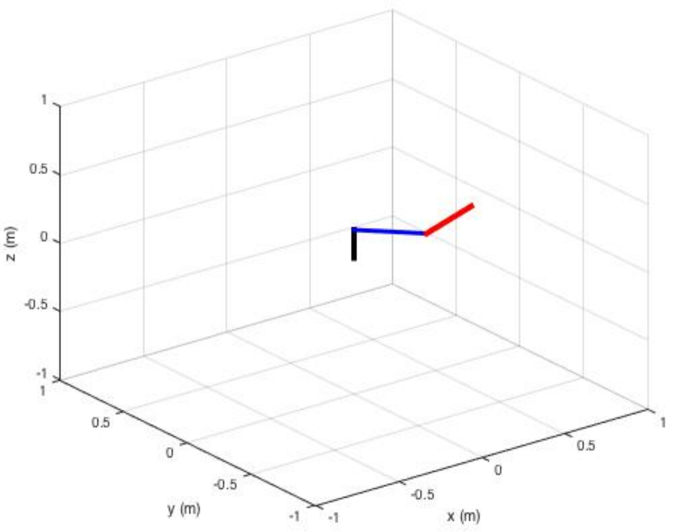


Figure 2.4: Manipulator Configuration With Random Angles

2.2.1 The Jacobian Matrix

The Jacobian matrix is a $6xn$ matrix, where n is the number of joints. It is one of the most useful tools in manipulator characterization since it is used in finding singularities (configurations where the manipulator uses one or more degrees of freedom), determining inverse kinematics algorithms, deriving dynamic equations of motions and many other functions. Like the homogeneous transformation matrix conveys the position and orientation of the end-effector, the Jacobian matrix describes how they are affected with changes in the robot parameters. For instance, column i of the Jacobian represents the contribution of the linear and angular velocity changes of joint i in the change of the end-effector's pose. In other words, it is used to express the end-effector's linear and angular velocities as a function of the joint velocities. The Jacobian matrix can be obtained either analytically or geometrically.

Analytic Jacobian The analytic Jacobian $J_a(q)$ is computed by differentiating the end-effector's pose function 2.4 obtained in the forward kinematics algorithm with respect to the joint variables and thus obtaining a $6x6$ Jacobian matrix.

The analytical technique is represented by first differntiating the end-effector's position vector with respect to the joint variables to form the first three rows of the Jacobian (2.14) and then doing the same with the end-effector's orientation to form the last three rows (2.15). Consequently, the analytical Jacobian J_a is fomred using (2.16).

$$\dot{p}_e = J_P(q)\dot{q} \tag{2.14}$$

$$\dot{\phi}_e = J_\phi(q)\dot{q} \tag{2.15}$$

$$J_a = \begin{bmatrix} J_P \\ J_\phi \end{bmatrix} \tag{2.16}$$

Geometric Jacobian The geometric Jacobian $J(q)$ is calculated by first obtaining the linear and angular velocities (using 2.17 and 2.18) of each joint in terms of the joint angles $q_1, q_2, ..., q_n$ and then isolating the coefficients of $\dot{q}$ and forming the Jacobian.

$$\dot{p}_e = J_P(q)\dot{q} \tag{2.17}$$

$$w_e = J_O(q)\dot{q} \tag{2.18}$$

$$v_e = \begin{bmatrix} \dot{p}_e \\ w_e \end{bmatrix} = J(q)\dot{q} \tag{2.19}$$

2.19 represents the manipulator's differential kinematics, where $J(q)$ is the $6x6$ geometric Jacobian of the arm.
In order to compute the angular and linear velocities of each joint in terms of the joint variables, formulas 2.20 and 2.21 are utilized.

$$^{i+1}\omega_{i+1} =_i^{i+1} R.^i\omega_i + qi + 1^{i+1}\dot{Z}_{i+1}(2.20)$$

$$^{i+1}v_{i+1} =_i^{i+1} R(^iv_i +^i \omega_i X^i P_{i+1}) \tag{2.21}$$

By collecting the factors of terms that are multiples of $\dot{q_1}, \dot{q_2}, ..and\dot{q_6}$ the elements of the geometric Jacobian are obtained.

2.2.2 Inverse Kinematics Iterative Method

To solve the inverse kinematics problem iteratively the Gradient Descent method shown in 2.23 is used. First initial random joint angles are chosen. Then the iterations in 2.23 are implemented until the Euclidean norm of the error 2.22 is less than a set threshold. The pseudo-inverse of the Jacobian can be used instead of the Jacobian transpose. The pseudo-inverse is usually used to overcome the problem of a singular Jacobian or a non-square Jacobian. However, since the pseudo-inverse is not working for this Jacobian, the transpose is used.

$$e(q_k) = k_d - k(q_k) \tag{2.22}$$

$$q_{k+1} = q_k + \alpha J_a^T(k_d - k(q_k)) \tag{2.23}$$

In this iterative method, a threshold of 0.001 is set and the Jacobian utilized is the analytic Jacobian J_a. Furthermore, to choose the constant α, several values are tested. The plot in figure 2.5 illustrates the time to convergence of the tested values. As shown, when setting $\alpha = 6.2$ the algorithm converges faster obtaining the proper joint angles to reach the desired pose.

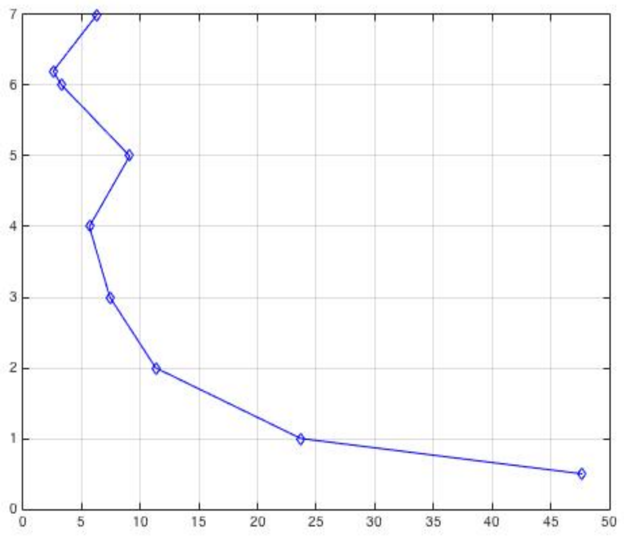


Figure 2.5: Convergence time of different values of α

As mentioned before, the end-effector's pose is represented by the position and orientation of the end-effector. In this study, however, the orientation is neglected and

only the position part is solved. The reason is that the inverse kinematics iterative method does not converge when testing the whole pose but it does when testing the position only. Therefore the Jacobian utilized is the $3x6$ analytic Jacobian J_P, which is determined by differentiating the position vector P_e with respect to the joint variables, i.e. the first three rows of J_a.
As an example, the robot configuration, i.e. the suitable joint angles, for a random desired position (2.24) is required. After choosing any random initial angles and setting $\alpha = 6.2$ the iteration 2.23 is implemented until ther error is less than 0.001. 2.25 shows the last value of q_k which is the vector of the essential joint angles that are needed to reach the desired position k_d.

$$k_d = \begin{bmatrix} 0.4 \\ 0.2 \\ 0.3 \end{bmatrix} \tag{2.24}$$

$$q_k = \begin{bmatrix} 1.11 \\ 0.634 \\ 1.6 \\ 1.0 \\ 0 \\ 0.5 \end{bmatrix} \tag{2.25}$$

To test the inverse kinematics algorithm the obtained joint angles are tested with the forward kinematics algorithm. Figures 2.6 and 2.7 illustrate the manipulator's configuration using the resulting joint angles. The arrows point to the position of the end-effector which matches the desired position in 2.24.

2.2.3 Joint Space Trajectories for Point-to-Point Motion

Now, it is required to design joint space trajectories for point-to-point motion in the task space. The joint space is the set of all possible joint parameters and the task space is the set of all possible poses of the end-effector. A point-to-point motion is where the manipulator moves from an initial configuration q_i to a final configuration q_f in a specified time t_f. The time is to be chosen carefully in order not to exceed the joints' dynamic limits. The trajectory scaling method is a method that can be used to make sure the required motion time does not violate the dynamic limits. Here, the time t_f specified will be chosen randomly to solve the example without taking the torque limits into consideration.

Using polynomials is a natural choice to design joint trajectories for a point to point motion, since they provide smooth continuous motion. A cubic polynomial as in 2.26 is

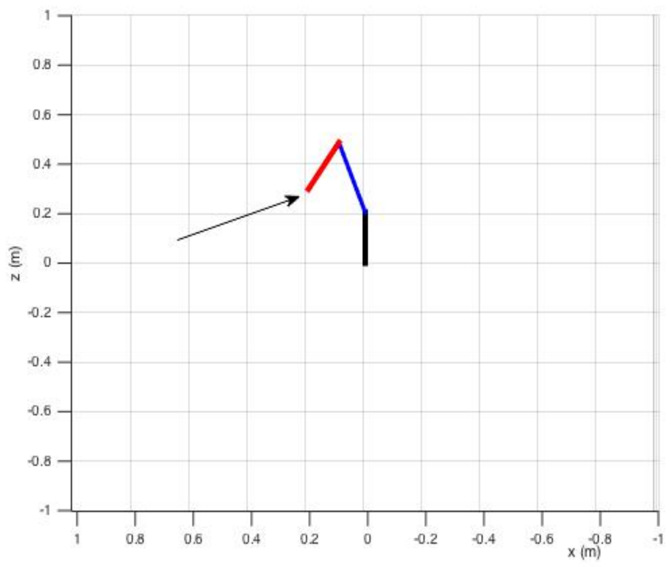


Figure 2.6: End-effector Position in X-Z Plane

used.

$$q(t) = a_3t^3 + a_2t^2 + a_1t + a_0 \tag{2.26}$$

2.26 is differentiated with respect to time twice in order to get the velocity as well as the acceleration profile (2.27 and 2.28).

$$\dot{q}(t) = 3a_3t^2 + 2a_2t + a_1 \tag{2.27}$$

$$\ddot{q}(t) = 6a_3t + 2a_2 \tag{2.28}$$

To determine a specific trajectory, the initial time t_i is set to zero and the following system of equation shall be solved.

$$a_0 = q_i \tag{2.29}$$

$$a_1 = \dot{q}_i \tag{2.30}$$

$$q_f = a_3t_f^3 + a_2t_f^2 + a_1t_f + a_0 \tag{2.31}$$

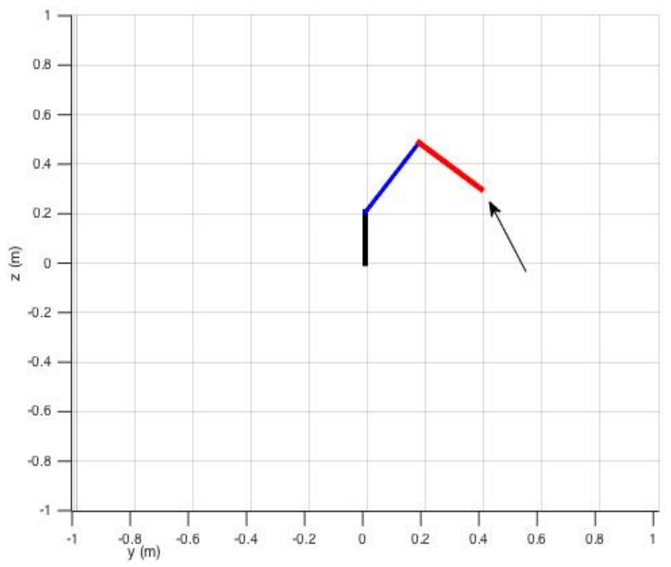


Figure 2.7: End-effector Position in Y-Z Plane

$$\dot{q}_f = 3a_3t_f^2 + 2a_2t_f + a_1 \tag{2.32}$$

Given the initial and final set of joint angles, q_i and q_f, by setting the initial and final velocities, $\dot{q}_i$ and $\dot{q}_f$ to zero, and specifying the time t_f the coefficients of the cubic polynomial in 2.26 are determined.

The example from the inverse kinematics part will be used to explain how this method works. The starting configuration is the manipulator zero configuration where the position of the end-effector is $p_e = [000.86]^T$ and it goes to the final configuration which is shown in 2.25. First the four coefficients a_0, a_1, a_2 and a_3 are computed using 2.29 to 2.32. The determined coefficients are then substituted into equations 2.26, 2.27 and 2.28 to obtain the position, velocity and acceleration profile, respectively. Figures 2.8, 2.9 and 2.10 illustrate the position, velocity and acceleration profiles in an infinite time. The profiles as expected:

- Position profile: cubic function
- Velocity profile: parabolic function
- Acceleration profile: linear function

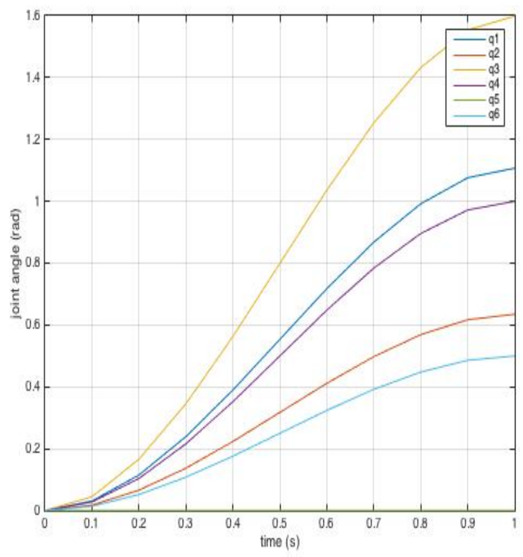


Figure 2.8: Position Profile in a Cubic Polynomial Timing Law

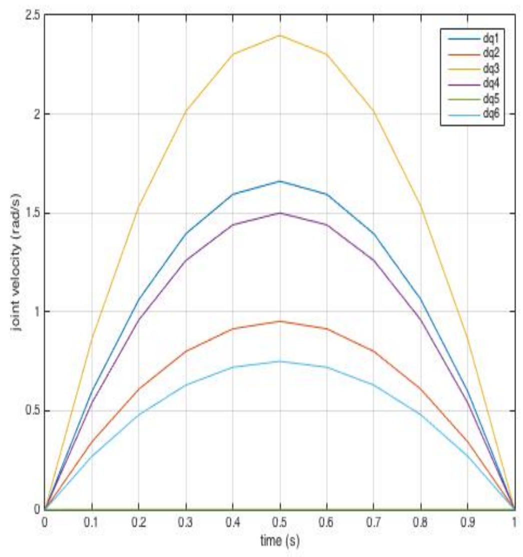


Figure 2.9: Velocity Profile in a Cubic Polynomial Timing Law

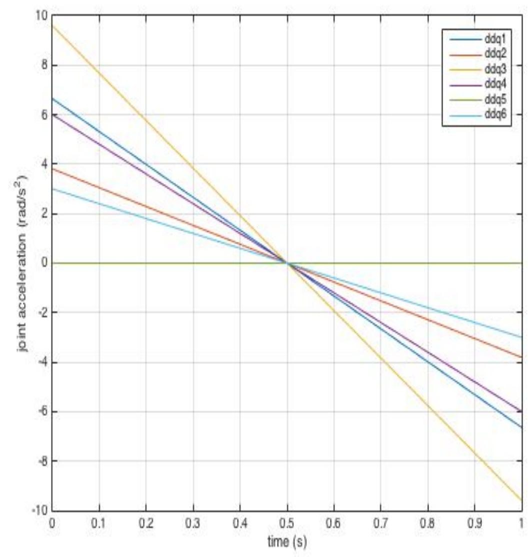


Figure 2.10: Acceleration Profile in a Cubic Polynomial Timing Law

Another type of polynomial that can be used to design the joint space trajectories is a fifth order polynomial (2.33). One advantage of using a fifth order polynomial is the ability of assigning initial and final values for the acceleration as well. In that case six constraints have to be satisfied. The coefficients a_0 to a_5 are computed as for the third order polynomial and then the trajectory can be easily designed.

$$q(t) = a_5 t^5 + a_4 t^4 + a_3 t^3 + a_2 t^2 + a_1 t + a_0 \tag{2.33}$$

2.3 Kinematic Control

Author: *Daniel Heinze*

Precise control of the modular robot requires the development of an kinematic control algorithm, which takes the desired position and angles as an input and interpolates the movement of the robots joints in time according to its starting and required final position.

2.3.1 Robot Assemblies

In order to derive the DH-tables and later the transformation matrix, we need to create assemblies of the robot. Additionally we need to specify the x and z axis, to set a consistent orientation across the whole project and teams.
Figure 2.11 contains the two assemblies with all x and z axis.

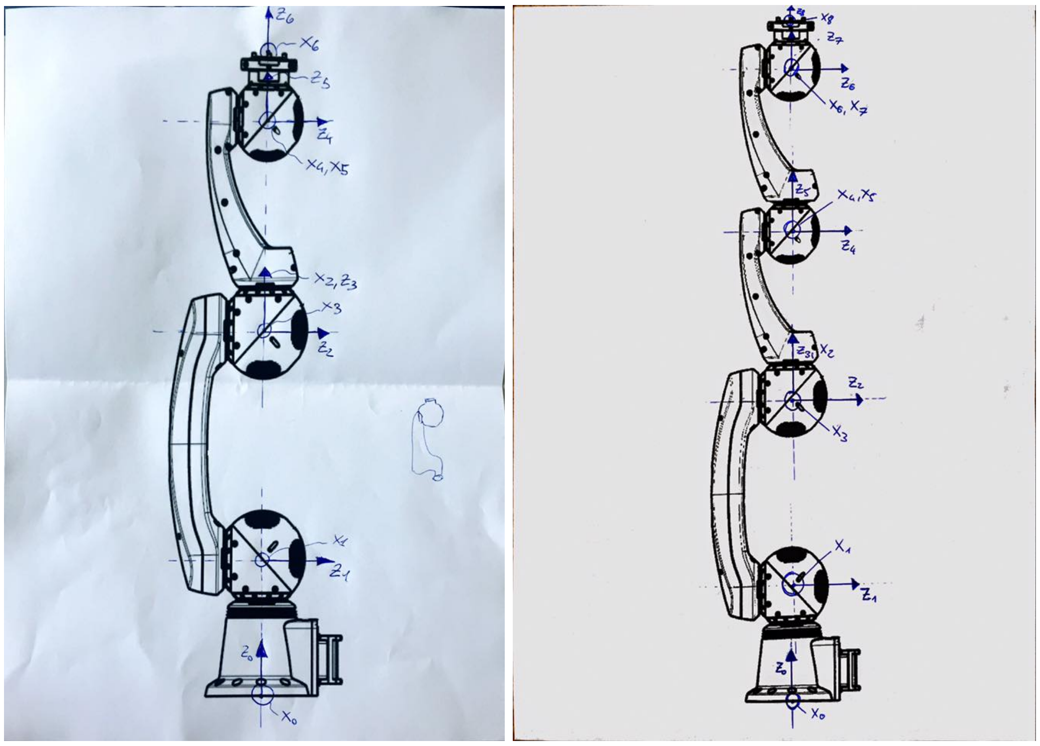


Figure 2.11: Left: Assembly variant 1. All calculations are based on this.
Right: Assembly variant 2.

The z-axis indicates the axis around which this specific joint can move.

2.4 Kinematic Control in task space

The kinematic control describes the movement of the robot with x,y and z coordinates and its yaw, pitch and roll angles. In order to achieve this movement the trajectories of the robot have to be designed and a kinematic control algorithm needs to be found. The findings of the previous chapters, like the Jacobian or the Forward-Kinematics algorithm will help achieving this goal.

2.4.1 Trajectory design in task space

The trajectory algorithm uses two 6x1 vectors as an input (which contains x,y,z and the 3 angles). These represent the start point and desired point of movement, where the end-effector of the robot should move to. Since the algorithm is time-dependent, which allows to get the position of the end-effector after every time step, the algorithm also needs to take the start and end time as an input.
Siciliano et al. [SSVO10] describe, how movement along one dimension can be interpolated by using quintic polynomials. Quintic polynomials are polynomials of 5th order. These polynomials in general are given in equation (2.34)

$$f(x) = a_0x^5 + a_1x^4 + a_2x^3 + a_3x^2 + a_4x + a_5 \tag{2.34}$$

This leads to a linear equation system A*x = b , where b is a 6x1 vector with initial and final position, velocity and acceleration, x are our desired parameters of the quintic polynomial and A is the matrix of the polynomial values. This matrix is shown in equation (2.35):

$$A = \begin{pmatrix} 1 & t_0 & t_0^2 & t_0^3 & t_0^4 & t_0^5 \\ 0 & 1 & 2*t_0 & 3*t_0^2 & 4*t_0^3 & 5*t_0^4 \\ 0 & 0 & 2 & 6*t_0 & 12*t_0^2 & 20*t_0^3 \\ 1 & t_f & t_f^2 & t_f^3 & t_f^4 & t_f^5 \\ 0 & 1 & 2*t_f & 3*t_f^2 & 4*t_f^3 & 5*t_f^4 \\ 0 & 0 & 2 & 6*t_f & 12*t_f^2 & 20*t_f^3 \end{pmatrix}$$

To test the behaviour we use the following Simulink model. (See figure 2.12)

An example for x,y and z leads to the output (x-axis = time) in figure 2.13

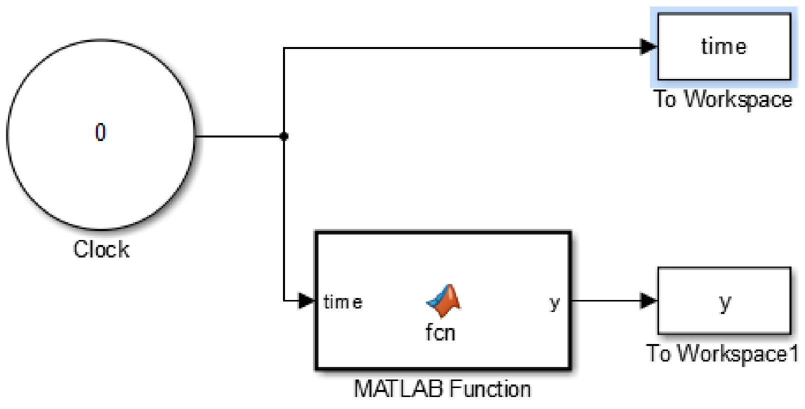


Figure 2.12: Simulink model for the trajectory design.

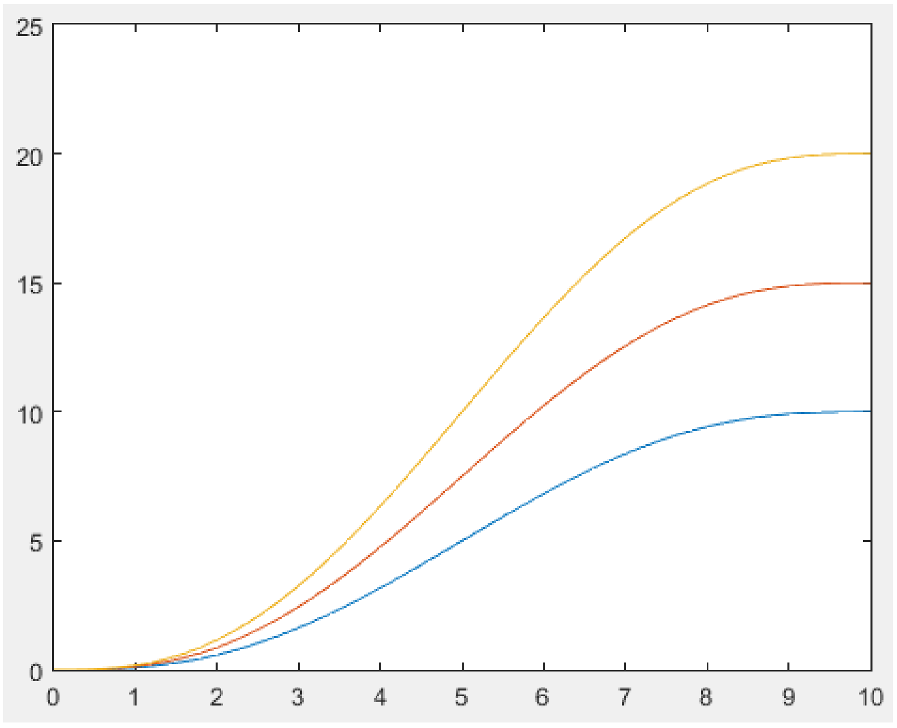
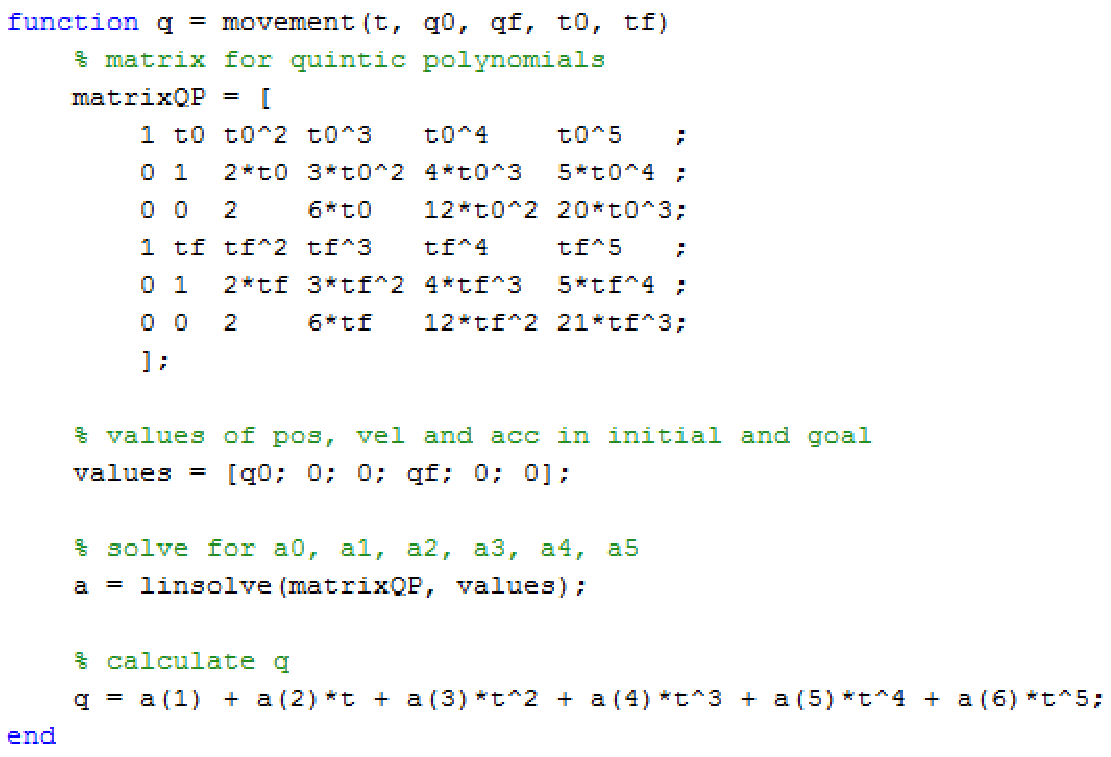


Figure 2.13: Example output of the designed trajectory algorithm.



```
function q = movement(t, q0, qf, t0, tf)
    % matrix for quintic polynomials
    matrixQP = [
        1 t0 t0^2 t0^3   t0^4    t0^5   ;
        0 1  2*t0 3*t0^2 4*t0^3  5*t0^4 ;
        0 0  2    6*t0   12*t0^2 20*t0^3;
        1 tf tf^2 tf^3   tf^4    tf^5   ;
        0 1  2*tf 3*tf^2 4*tf^3  5*tf^4 ;
        0 0  2    6*tf   12*tf^2 21*tf^3;
        ];

    % values of pos, vel and acc in initial and goal
    values = [q0; 0; 0; qf; 0; 0];

    % solve for a0, a1, a2, a3, a4, a5
    a = linsolve(matrixQP, values);

    % calculate q
    q = a(1) + a(2)*t + a(3)*t^2 + a(4)*t^3 + a(5)*t^4 + a(6)*t^5;
end
```

Figure 2.14: Matlab code for the trajectory design.

2.4.2 Kinematic control

In this chapter the development of an kinematic control algorithm is described. This follows the approaches by Siciliano et al. [SSVO10] and Spong et al. [SHV06].

An kinematic control algorithm is used, to get the movement of each joint from the robot to transform the end-effector to a desired point. It takes a 6-dimesional vector as input (position: 3, angles: jaw, pitch, roll) and converts them into a q vector with movement values for each joint. This is done for each time step.

The Simulink model for the kinematic control is displayed in figure 2.15

The model takes the desired position (6x1 vector) and its derivative as input and outputs the current position of the system. It then subtracts the input by x_e (position of end effector) to minimize the error of the system. After adding a gain of 100* I which showed to be a efficient constant to use for this algorithm setup, we use (2.36) to

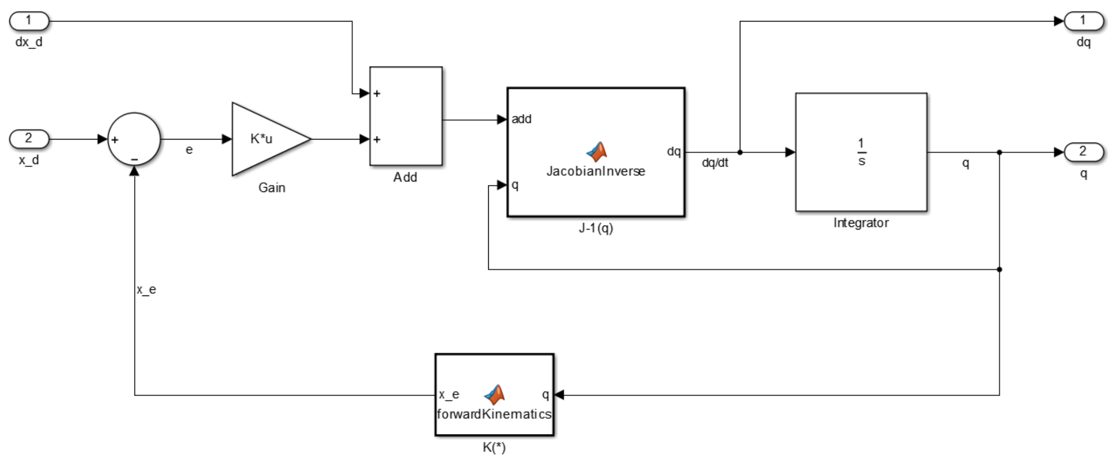


Figure 2.15: Simulink model for kinematic control.

calculate dq/dt.

$$\dot{q} = J_A^T(q) * (\dot{x_d} + K * e) \tag{2.36}$$

In this equation the transpose Jacobian is used, which contains the information of all robot joints. It is possible to use the inverse Jacobian in the equation, but this proved to be extremely inefficient in test of the algorithm.

After this calculation we use the Simulink integrator to get the 6-dimensional q vector. The initial values given in equation (2.37) are used, to calculate outputs, before the actual input is used. To minimize the error of the system, the forward kinematics algorithm is used, to calculate the movement of the end effector, which is subtracted from the input.

$$x_i = \begin{pmatrix} 0.5 \\ 1 \\ 1.5 \\ 1 \\ 0 \\ 0.5 \end{pmatrix}$$

In the following example it is shown, that the error minimizes after a short time after starting the system. The desired input is given in (2.38)

$$x_d = \begin{pmatrix} 0 \\ 0 \\ 0.3 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

After running the model the resulting diagram 2.16 shows the position of the end-effector in time. It can be clearly seen that shortly after the start each value is near to its desired input. The yellow line is the third row value of the vector which represents the z position and evaluates to 0.3.

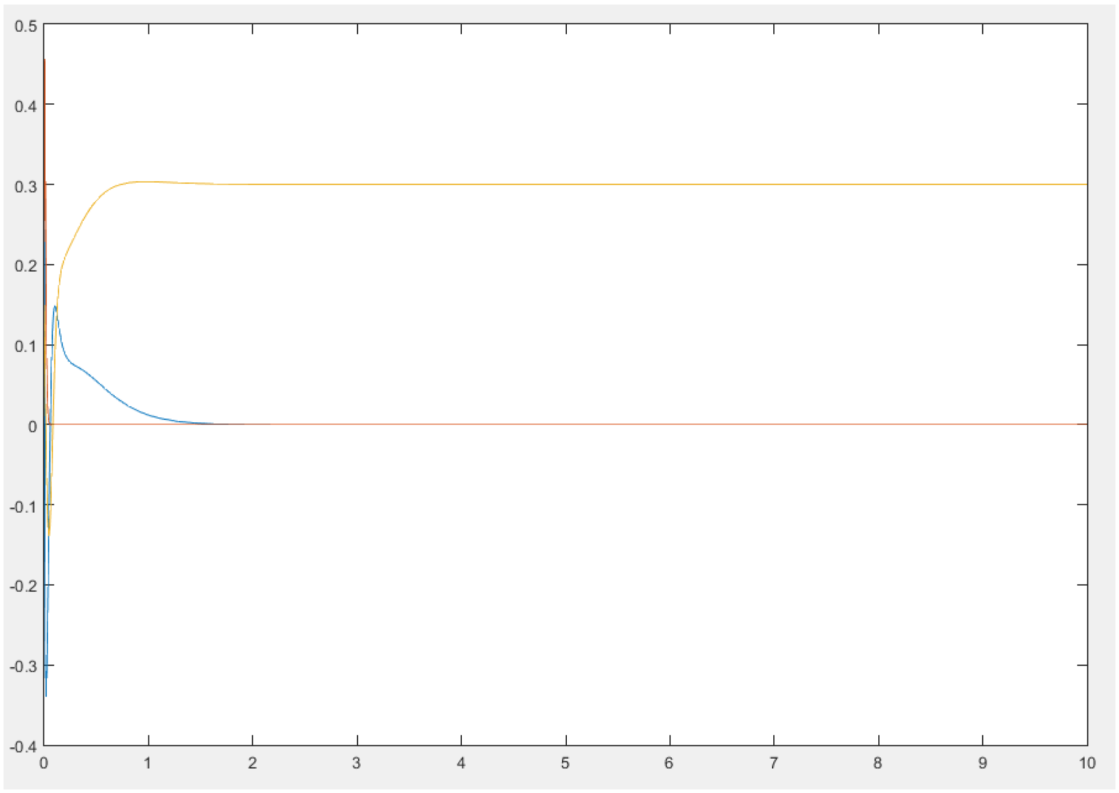


Figure 2.16: Example output of the kinematic control algorithm.

Further examples of outputs from the algorithm are:

$$x_d = \begin{pmatrix} 0 \\ 0.3 \\ 0.3 \\ 0.3 \\ 0.6 \\ 0.9 \end{pmatrix}$$

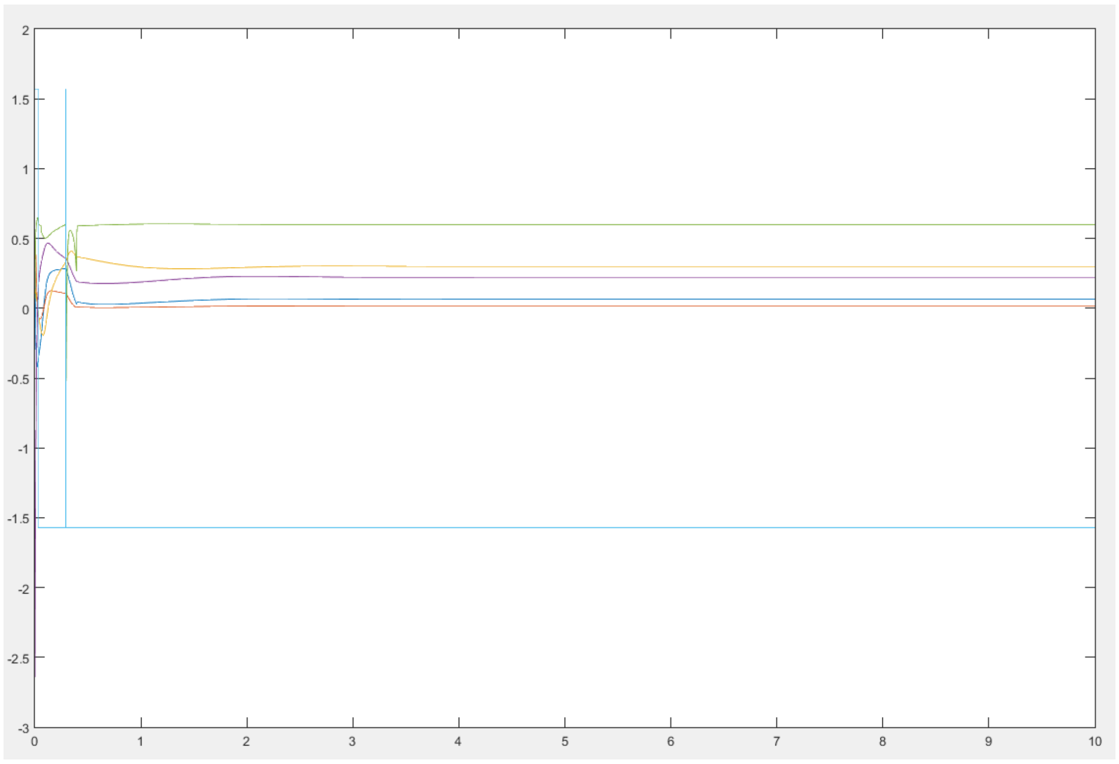


Figure 2.17: Second sample output of the kinematic control algorithm. Based on input (2.39)

$$x_d = \begin{pmatrix} 0.9 \\ 0.6 \\ 0.3 \\ 0.9 \\ 0.6 \\ 0.3 \end{pmatrix}$$

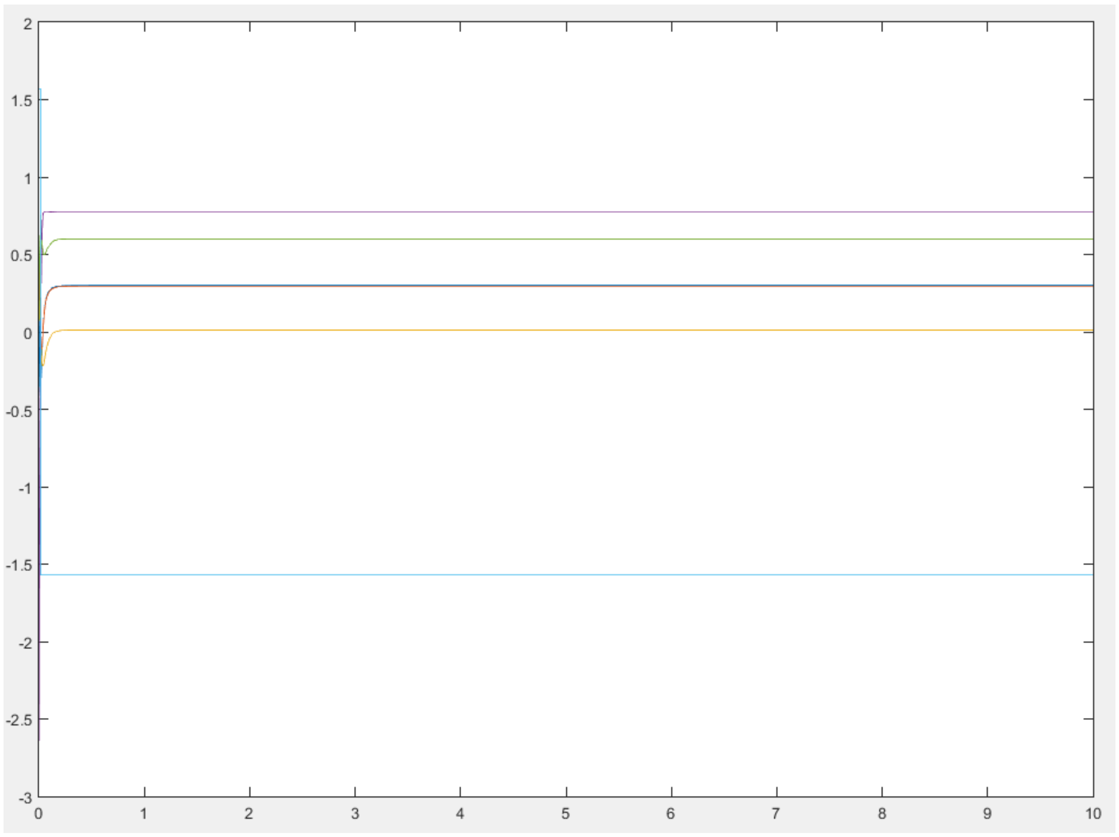


Figure 2.18: Third sample output of the kinematic control algorithm. Based on input (2.40)

3 Dynamical Model

Responsible: *Sophie Sepp and Benedikt Feldotto*

3.1 Equation of motion (EoM) of the manipulator

3.1.1 EoM using Newton-Euler Algorithm

There exist two different approaches for calculating the Equation of Motion for a robot. The Lagrangian method uses assumptions about the potential and kinetic energy in the various robot modules. Newton Euler's method calculates velocities and forces in a recursive way. Since Newton Eulers approach does not need any derivates, but Lagrangians does, calculating velocities and forces is computationally more effective. Because of this less complex and faster computation we will use the Newton Euler method to calculate the Equation of Motion for the robot. Afterwards it is used to build up a simulator of the robot in Matlab Simulink. Before we can start we have to adjust the coordinate frame, offered by the kinematics group, for our purpose.

Adjusting coordinate frames

For the kinematic considerations the coordinate frames where attached as simple as possible, in this case two at each center of the power balls. For the dynamic calculation nevertheless we need to adjust the coordinate frames to the end of each link. In this coordinate frames also the center of mass as well as the inertia are described. We transform the coordinate systems to the desired positions as seen in Figure 3.1 and calculate the adjustment with regular transition vectors and rotation matrices.

The Inertia matrices now should be described in these coordinate systems. This is already done for the power-balls regarding the data sheets, for the extended links we use Equation 3.1 and Equation 3.6 to transform the matrices in another frame. In a last step we write the center of mass in our new coordinate frames.

Rotation of Inertia matrix:

$$I' = R * I * R^T \tag{3.1}$$

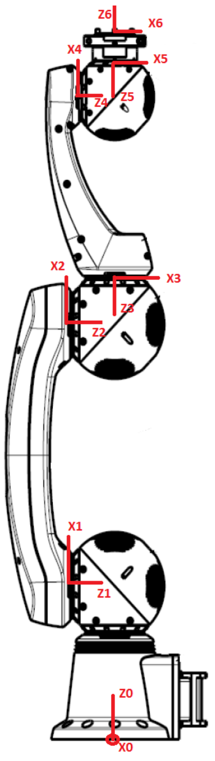


Figure 3.1: adjusted coordinate frames on the robot

Translation of Inertia matrix:

$$I' = I * I_T \tag{3.2}$$
$$I_T = m * r_0^T * r_0 * 1 - m * r_0 * r_0^T \tag{3.3}$$

with:
I – original inertia matrix
I'– transformed inertia matrix
I_T– Inertia matrix for translation
r_o– translation vector

Newton Euler method – Inverse dynamics

The recursive Newton Euler method computes velocities and accelerations in a forward, forces and torques in a backward phase. Since our robot is fixed on a table and only gravitational force is applied to it, we can make some assumptions for the base of the robot [Bur15]:

$$
\begin{aligned}
{}^{0}\dot{v}_0 &= [0,0,-9.81]^T && (3.4)\\
{}^{0}w_0 &= [0,0,0]^T && (3.5)\\
{}^{0}\dot{w}_0 &= [0,0,0]^T && (3.6)
\end{aligned}
$$

Here the first index names the link frame in which values are expressed, the second one states the value corresponding link; all parameters are written in the corresponding 3D coordinate system using the [X,Y,Z] notation. In an iterative way now the velocities and acceleration of the following links can be computed. As the next link is also moved by movement of all the previous ones, previous values are transformed into the next link frame with the corresponding rotation matrix and added to the own link movements. For each link we us four formulas to calculate every link up to the tip:

$$
\begin{aligned}
{}^{i+1}\omega_{i+1} &= {}^{i+1}_{i}R \cdot {}^{i}\omega_i + \dot{\Theta}_{i+1} \cdot {}^{i+1}Z_{i+1}\\
{}^{i+1}\dot{\omega}_{i+1} &= {}^{i+1}_{i}R \cdot {}^{i}\dot{\omega}_i + {}^{i+1}_{i}R \cdot {}^{i}\omega_i \times \dot{\Theta}_{i+1} \cdot {}^{i+1}Z_{i+1} + \ddot{\Theta}_{i+1}{}^{i+1}Z_{i+1}\\
{}^{i+1}\dot{v}_{i+1} &= {}^{i+1}_{i}R({}^{i}\dot{\omega}_i \times {}^{i}P_{i+1} + {}^{i}\omega_i \times ({}^{i}\omega_i \times {}^{i}P_{i+1}) + {}^{i}\dot{v}_i)\\
{}^{i}\dot{v}_{C_i} &= {}^{i}\dot{\omega}_i \times {}^{i}P_{C_i} + {}^{i}\omega_i \times ({}^{i}\omega_i \times {}^{i}P_{C_i}) + {}^{i}\dot{v}_i
\end{aligned}
$$

Figure 3.2: Newton-Euler: velocities and accelerations

Forces and torques resulting of robot actuation are applied at the TIP but nevertheless will affect every joint of the robot. This is the reason why we now start at the TIP of the robot and calculate forces and torques for all links back to the base. For the general case we first assume that there are no external forces at the TIP and so:

$$
\begin{aligned}
{}^{6}f_6 &= [0,0,0]^T && (3.7)\\
{}^{6}n_6 &= [0,0,0]^T && (3.8)
\end{aligned}
$$

(for our Schunk LWA robot link number 6 is the last link, but since equations are equal for all links this can be extended as desired.) Each links forces and torques are calculated with the formulas in Figure 3.3 [Bur15]:

Our Robot only consists of rotational and no prismatic joints, so that the torque in each joint is the Z component of the torque vector of each link (Figure 3.4)

We implement the Newton Euler algorithm in a matlab function called "NE_rec". As an input we provide position q, velocity $\dot{q}$ and the acceleration $\ddot{q}$ as vectors with the

$$
\begin{aligned}
{}^{i}F_i &= m \cdot {}^{i}\dot{v}_{C_i} \\
{}^{i}N_i &= {}^{C_i}I_i \cdot {}^{i}\dot{\omega}_i + {}^{i}\omega_i \times {}^{C_i}I_i \cdot {}^{i}\omega_i \\
{}^{i}f_i &= {}^{i}_{i+1}R \cdot {}^{i+1}f_{i+1} + {}^{i}F_i \\
{}^{i}n_i &= {}^{i}N_i + {}^{i}_{i+1}R \cdot {}^{i+1}n_{i+1} + {}^{i}P_{C_i} \times {}^{i}F_i + {}^{i}P_{i+1} \times {}^{i}_{i+1}R^{i+1}f_{i+1}
\end{aligned}
$$

Figure 3.3: Newton-Euler: torques and forces

$$\tau_i = {}^{i}n_i^{\mathrm{T}} \cdot {}^{i}Z_i$$

Figure 3.4: Newton Euler: joint torque

elements of all joints and the vector for gravity in the base frame. Since the calculation is done in a forward and backward phase with itself equal formulas we implement two for-loops, besides the initialization for base and TIP. The algorithm provides torque values for all the joints and reshaped as a vector this is also the output of our function NE_rec.

Equation of Motion (EoM)

The Equation of Motion (Figure 3.5) describes the torques of the robots joints with Newton's Law. Additional terms for friction, centripetal force and gravity are added. To get our final equation we have to determine the mass matrix M and the vector n which includes centrifugal and centripetal forces C, friction v and gravity g.

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \overbrace{\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{v}(\dot{\mathbf{q}}) + \mathbf{g}(\mathbf{q})}^{\mathbf{n}(\mathbf{q},\dot{\mathbf{q}})} = \mathbf{u},$$

Figure 3.5: Equation of Motion

For this purpose we call the NE_rec function with specific parameters as shown in Figure 3.6 [Giu15] and save the results as new matlab functions so that we can call them later quickly to get the values. We use symbolic variables as place-holders for the robot configuration $q, \dot{q}, \ddot{q}$.

3.1.2 Simulating the robot

The calculated EoM fully describes the dynamic of the robot (except motor dynamics) and so can also be used to build up a simulator for evaluation purpose. The robot gets

- $\mathbf{n}(\mathbf{q}, \dot{\mathbf{q}}) = NE_{rec}(\mathbf{q}, \dot{\mathbf{q}}, \mathbf{0}, \mathbf{g})$
- the i-th column of $\mathbf{M}(\mathbf{q})$ is obtained as $\mathbf{M}_i(\mathbf{q}) = NE_{rec}(\mathbf{q}, \mathbf{0}, \mathbf{k}_i, \mathbf{0})$, with $\mathbf{k}_i$ selected to get the i-th colum (e.g. for second column $\mathbf{k}_i = [0\ 1\ 0\ 0 \cdots\ 0]^T$)

Figure 3.6: NE_rec function call to get EoM parameters

the torque vector as an input and position, velocities and accelerations can be measured. For calculation of the acceleration from torque we reorder the Equation of Motion:

$$\ddot{q} = inv[M(q)] * [u - n(q, \dot{q})] \tag{3.9}$$

Guided by this equation we build up a Simulink model, copying the structure. We use the saved values for M and n to enable fast simulation. Integrating the acceleration two times we get velocity and position, respectively, which can themselves be used for calculation. Figure 3.7 shows the setup in the simulator subsystem.

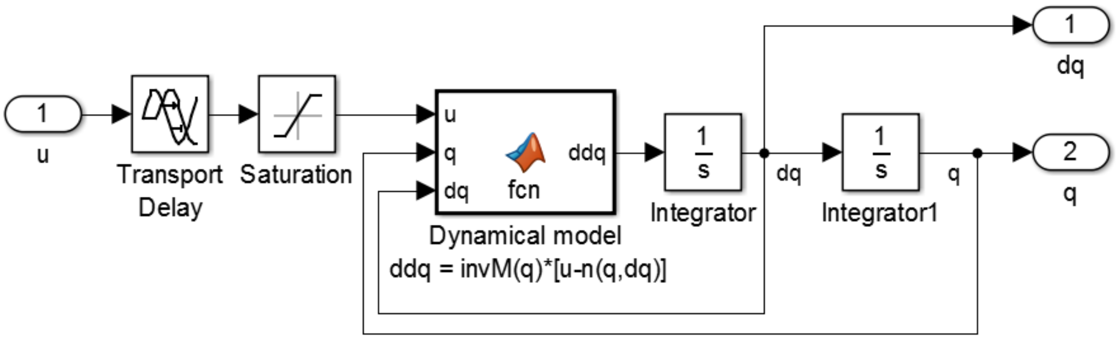


Figure 3.7: Robot dynamics simulation

3.2 Deriving a dynamical model of the actuators and combining it with the Simulink Model of task B.1)

[G.P12]. [M.N08]. [HKSR]. [MWS06].

In task B.2 a dynamical model of the actuators is derived and the Simulink model of task B.1 is enhanced with the actuator models. The robotic arm LWA 4P has six brushless DC motors which are modelled in Simulink to be combined with the Simulink model of task B.1. The modelling of the motor is hereby divided by the electrical part of the motor equations and the dynamical part of the motor equations.

3.2.1 Theoretical background: DC motor

A DC-motor works on the principle that a current-carrying conductor in a magnetic field experiences a force, which can be expressed as the product of magnetic flux and the current in the conductor. It consists of a fixed stator, which provides a constant magnetic field and a armature part, which is a rotor that rotates inside the stator and is a simple coil. The armature is connected to a DC power source through a pair of commutator rings. The current that flows through the coil induces a electromagnetic force on it according to the Lorentz force, which causes the coil to rotate. Thus the commutator rings connect with the power source of opposite polarity. As a result, on the left side electricity flows away while on the right side, electricity flows towards. Thus the torque action is in the same direction throughout the motion, which is the reason why the coil continues rotating. A characteristical aspect of DC motors is the production of back EMF. A rotating loop in a magnetic field produces an emf according to the principle of electromagnetic induction, which are in this case the armature loops. So an internal EMF will be induced, that opposes the applied input voltage.

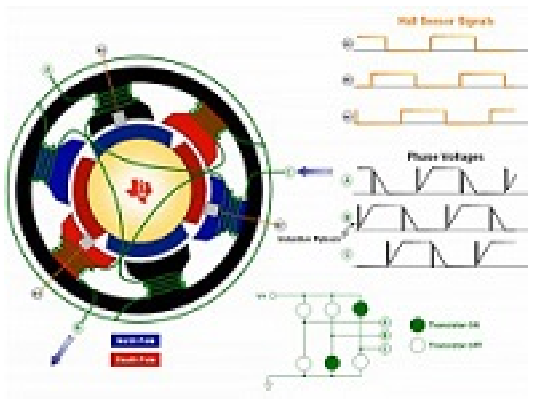


Figure 3.8: DC motor.

3.2.2 Equations describing DC motor

Electrical part of the motor equations:

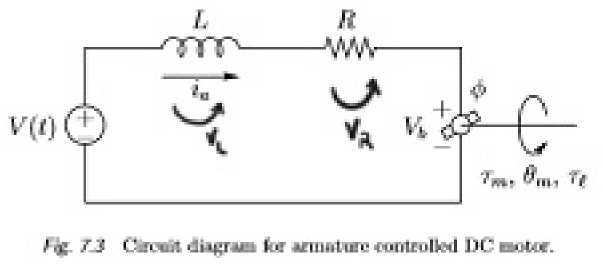

L
R
i_a
V_L
V_R
$V(t)$
+
−
V_b
ϕ
$\tau_m, \theta_m, \tau_\ell$
Fig. 7.3 Circuit diagram for armature controlled DC motor.

Figure 3.9: Circuit diagram for armature controlled DC motor.

V(t): Applied voltage
L: Armature self inductance
R: Armature resistence
K_b: Back emf constant
θ_m: mechanical rotor angle
τ_m: generated torque
τ_l: load torque

The voltage Vl and Vr can be computed in the following way:

$V_L = L \times \frac{di}{dt}$

$V_R = R \times i$

The electrical part of the motor can be computed by the sum of Vl, Vr and the back EMF.

$V_t(t) = L \times \frac{di}{dt} + R \times i(t) + V_b$

The back (induced) electromotive force V_b is proportional to the angular velocity $\omega(t)$ seen at the shaft. K_b, the back EMF constant, also depends on certain physical properties of the motor.

$V_b(t) = K_b \times \omega(t)$

The mechanical part of the motor equations is derived using Newton's law:

$J \times \frac{d\omega}{dt} = \sum_{i=1}^{n} \tau_i = u(t) - B \times \omega + K_s \times (q - \theta)$

The inertial load J times the derivative of angular rate equals the sum of all the torques about the motor shaft and can be expressed by the sum of the induced current i times the torque constant Km, the negative product of the back emf constant Kb and the angular velocity ω. and the torsional stiffness times the difference of angular postions of the rotor and the joint.

$\frac{d\omega}{dt} = \frac{u(t) - B \times \omega + K_s \times (q - \theta)}{J}$

Moreover, the torque u seen at the shaft of the motor is proportional to the current i and can be expressed as the product of the current i and the torque constant K_m, which is related to physical properties of the motor, such as magnetic field strength.

$u(t) = K_m \times i(t)$

Including the gear ratio

$\omega_m \times \tau_m = \omega_o \times \tau_o$

$\frac{\omega_m}{\omega_o} = \frac{\tau_o}{\tau_m} = K_\tau = 160$

$\omega_m = K_\tau \times \omega_o$

$\omega_m = \dot{\theta_m}$

$\omega_o = \dot{\theta_o}$

$J_m \times \ddot{\theta_m} = -B \times \dot{\theta_m} + u + f \times \tau_m$

$J_o \times \ddot{\theta_o} = f \times \tau_o$

$f \times \tau_m = \frac{f \times \tau_o}{K_\tau}$

$V_b = K_v \times \omega_m = K_v \times K_\tau \times \omega_o = K_{eq} \times \omega_o$

We get:

$J_m \times \ddot{\theta_m} = -B \times \dot{\theta_m} + u + \frac{J_o \times \ddot{\theta_o}}{K_\tau}$

$J_m \times K_\tau \times \ddot{\theta_m} = -B \times K_\tau \times \dot{\theta_m} + u + \frac{J_o \times \ddot{\theta_o}}{K_\tau}$

$J_m \times (K_\tau)^2 \times \ddot{\theta_m} = -B \times (K_\tau)^2 \times \dot{\theta_m} + K_\tau \times u$

$J_{eq} = J_m \times (K_\tau)^2$

$B_{eq} = B \times (K_\tau)^2$

$u_{eq} = K_{eq} \times i$

$u_{max} = K_{eq} \times i_{max}$

So we get K_{eq} by dividing the maximum torque by the maximum current.

For the first type of motor this is $\frac{19Nm}{3.9A} = 4.872\frac{Nm}{A}$.

For the second type of motor this is $\frac{64Nm}{18.5A} = 3.459\frac{Nm}{A}$.

Having computed $J_{eq}, B_{eq}, u_{eq}, K_{eq}$, the values can be integrated into a Simulink Model.

3.2.3 Deriving a Simulink Model for the DC motor equations

The equations describing the DC motor can be integrated into a Simulink Model which combines the electrical part and the mechanical part and computes the torque that is produced by each motor given the applied input voltage and the angular position as inputs.

model dc motor.jpg

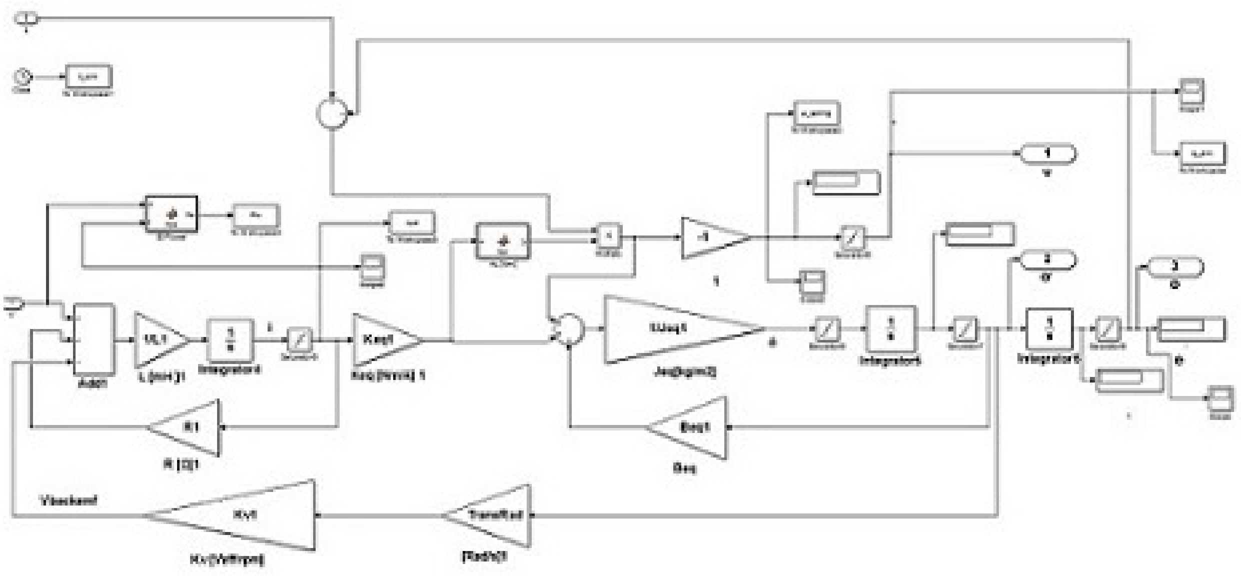


Figure 3.10: Simulink odel combining electrical and mechanical subsystem of DC motor.

The vector of the torques of all motors is given to the Simulink Model derived in task B.1 that describes the Equations of Motion, while the angular position that results from the Equations of Motion being applied is given back to the motor model.

combined.jpg

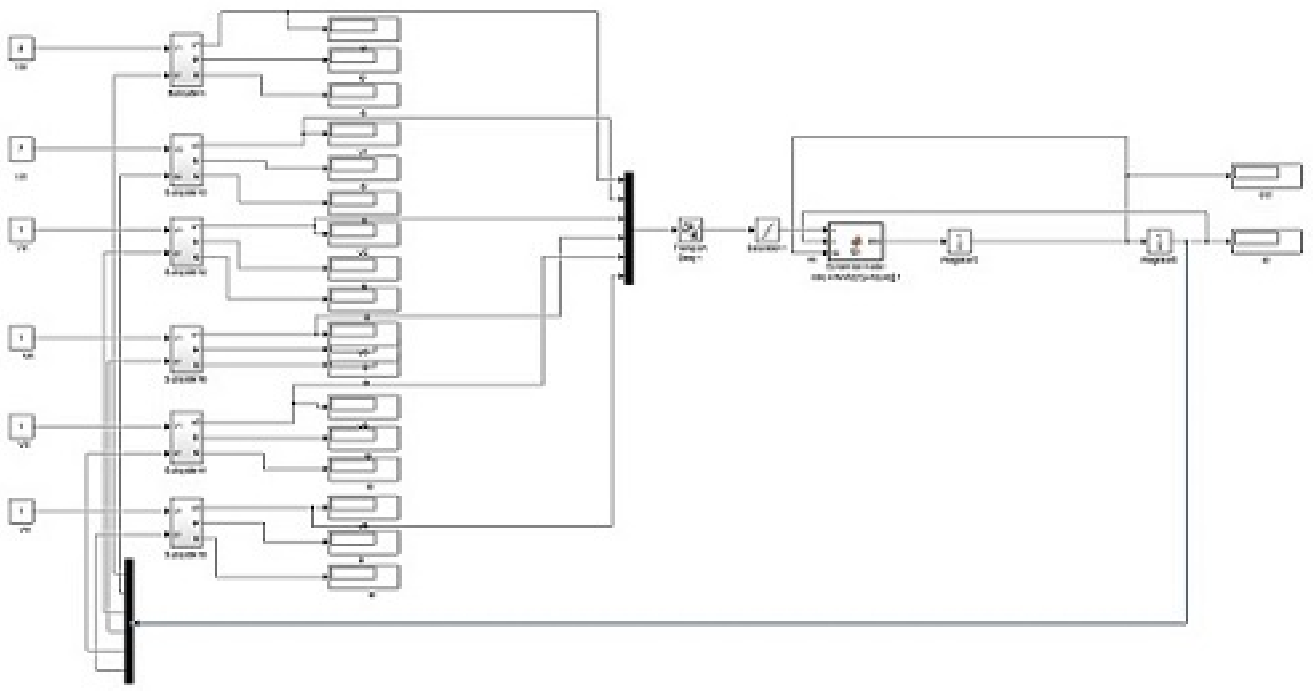


Figure 3.11: Combination of motor model with Newton Euler computation.

Comparison motor type 1 to motor type 2:

Both motors were run with the maximum input voltage of 24V for 10 seconds.

Electric Power (P)

power motor 1.jpg

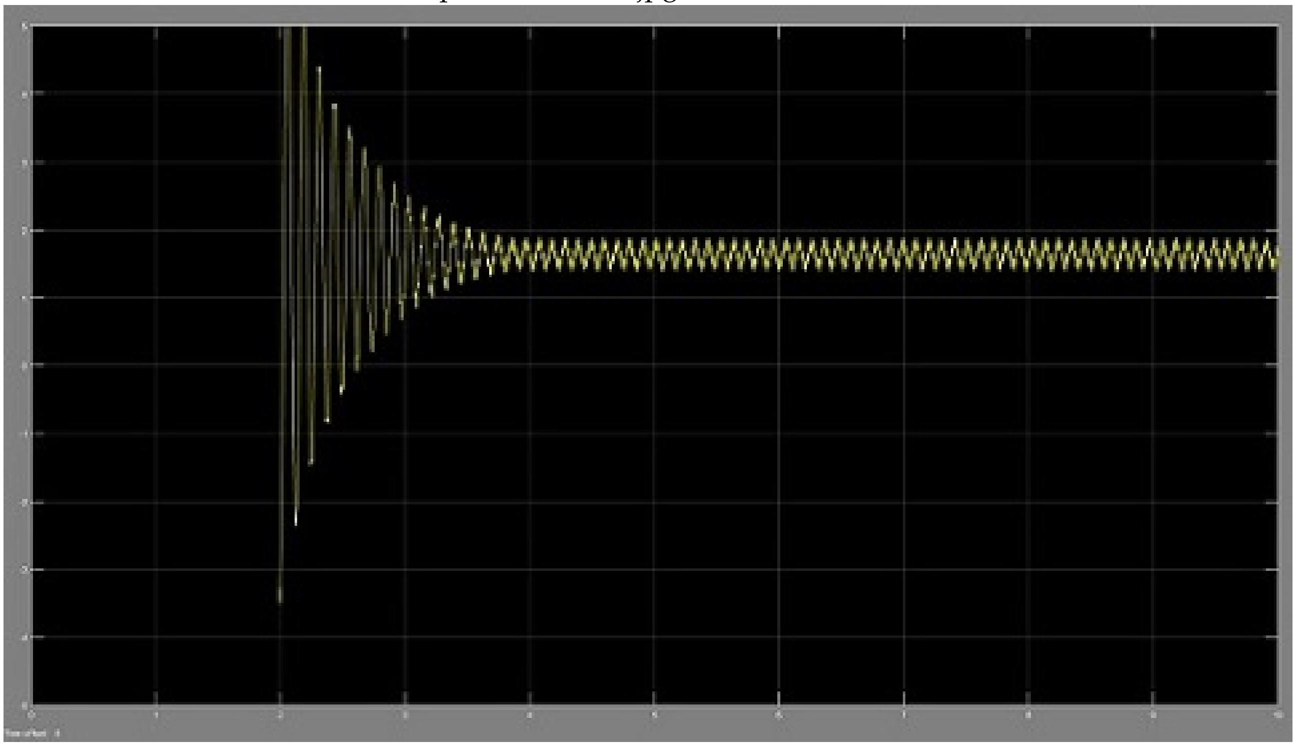


Figure 3.12: Electric power of motor type 1.

power motor 2.jpg

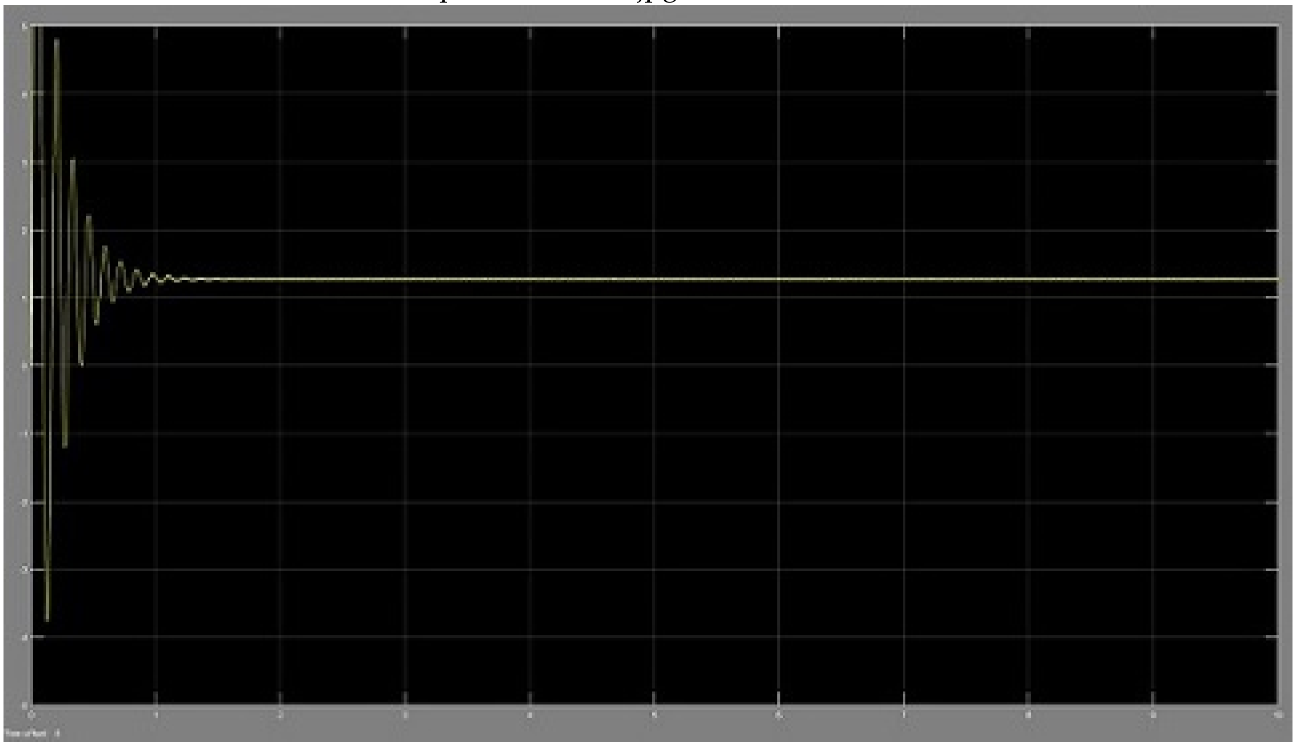


Figure 3.13: Electric power of motor type 2.

Induced Current (I/s)

current motor 1.jpg

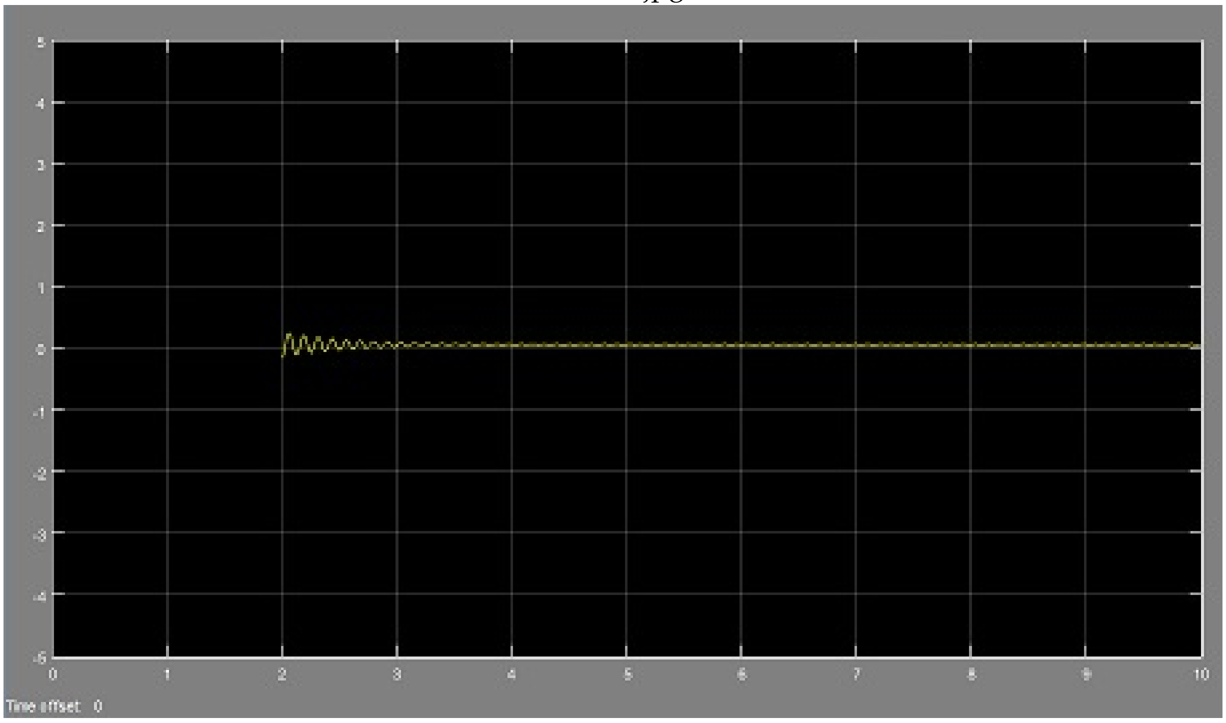


Figure 3.14: Induced current of motor type 1.

current motor 2.jpg

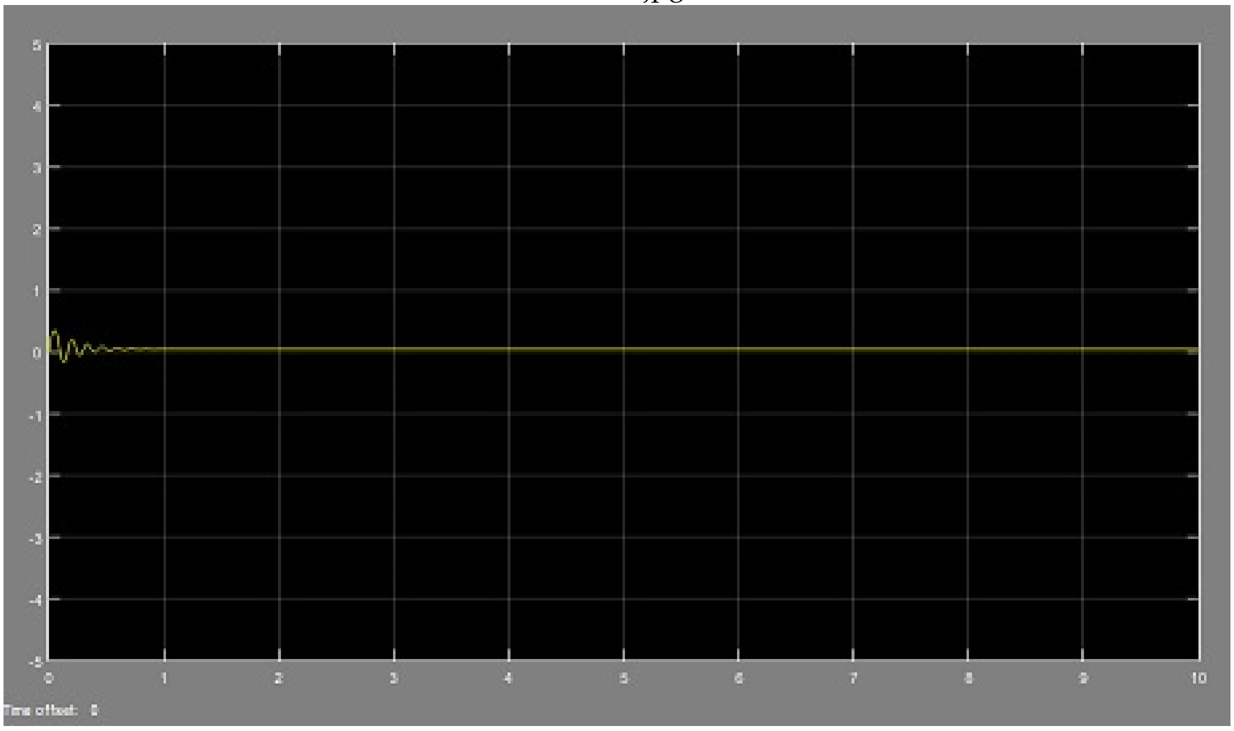


Figure 3.15: Induced current of motor type 2.

Torque (u/s)

motor 1.jpg

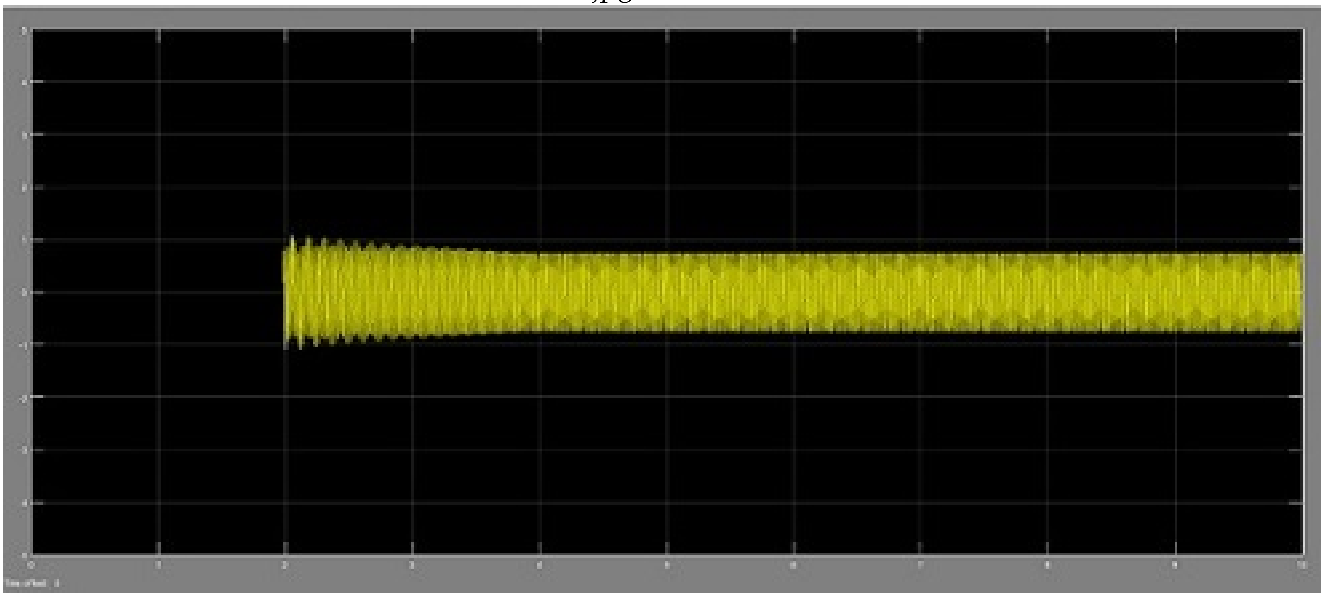


Figure 3.16: Torque of motor type 1.

motor 2.jpg

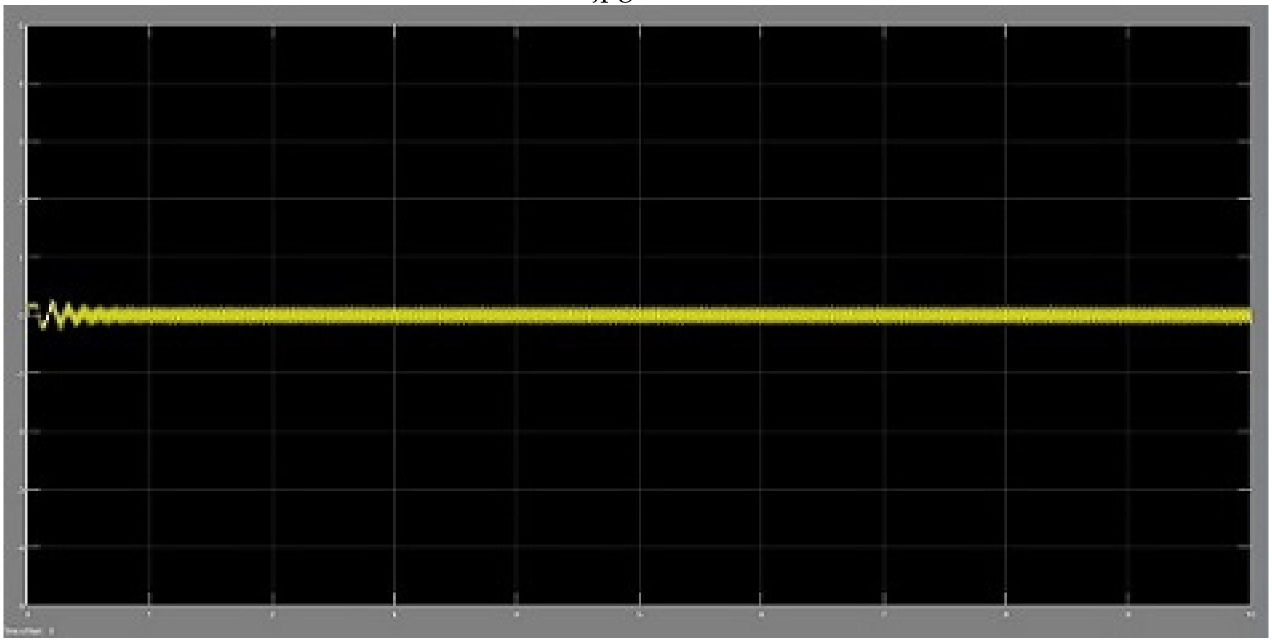


Figure 3.17: Torque of motor type 2.

Angular position (q/s)

position motor 1.jpg

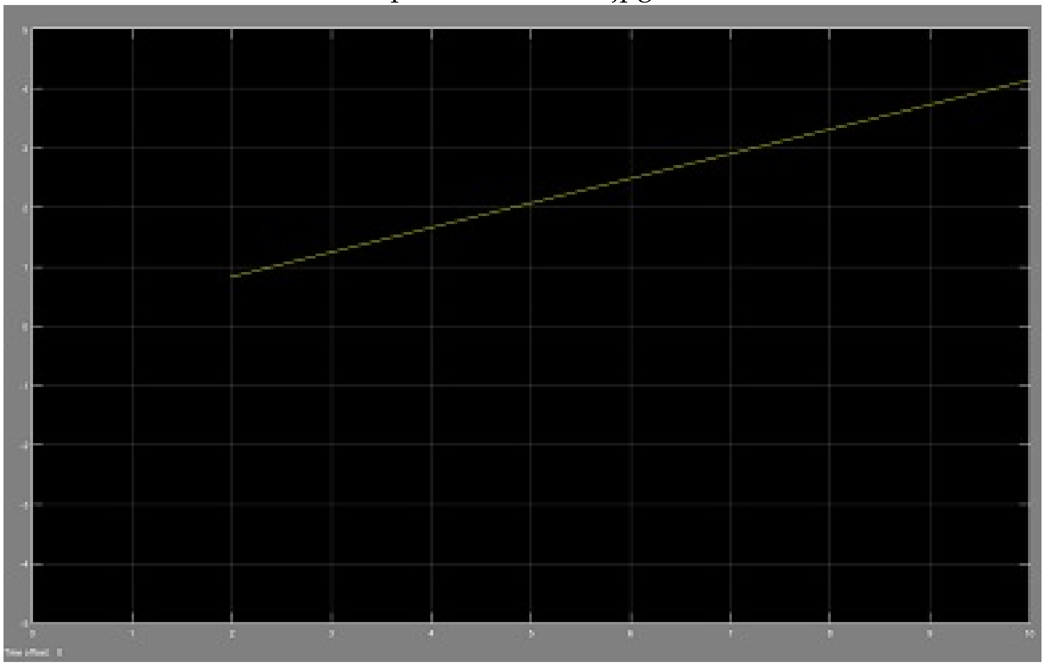


Figure 3.18: Angular position of motor type 1.

position motor 2.jpg

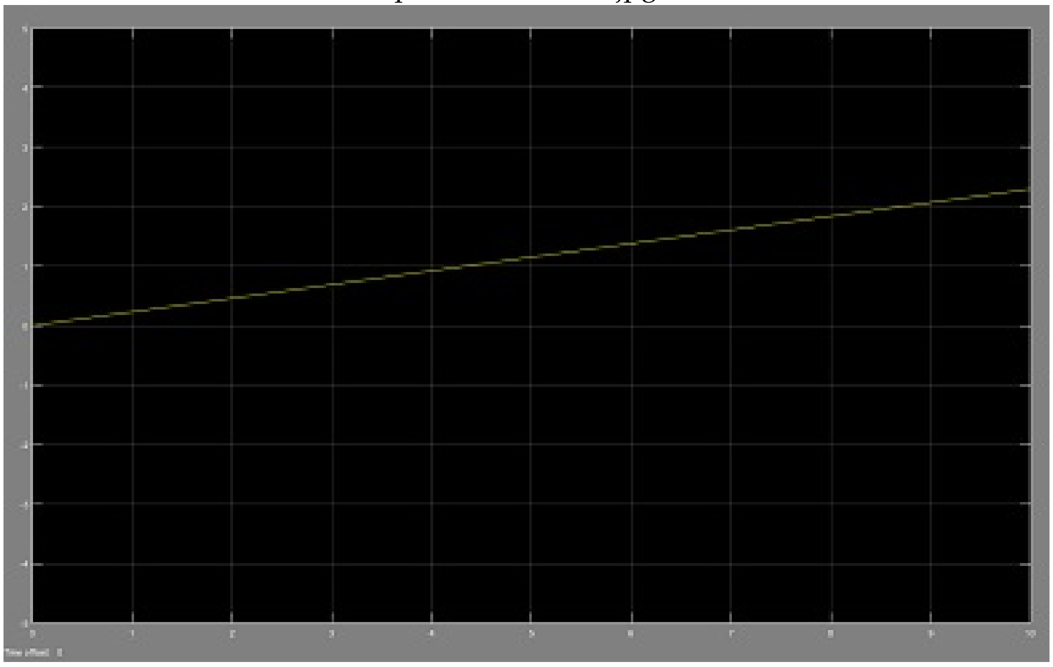


Figure 3.19: Angular position of motor type 2.

4 Task 3 - Joint-space controllers

Responsible: *Benedikt Feldotto*

4.1 Designing decentralized controllers

We now want to implement controllers for our robot. In our case control means reaching a desired position of the robot in short time, stable and robust, with as little overshooting as possible. For our modular robot approach we prefer first to build up a decentralized control structure. Each motor attached to the joints is controlled independently as a single-input/single-output system, so that the controller acts independently of the number of attached robot modules and no precise dynamic model of the robot is necessary. Getting a desired trajectory which includes desired positions (and eventually velocities and accelerations) of each link we can control the motors in joint-space. We set the goal angle for each motor individually and use the position sensors in the robot to get feedback about the actual state. This decoupled feedback control structure lets us build a robust controller.

Control effects

The motors are not able to bring the robot arbitrarily fast to the desired position. Inner effects such as self-induction and also all attached links influence the motor behaviour. The torque at the motors increase with each attached link as it is calculated with $\tau = F * s$. Partially not exactly calculable effects such as joint Coulomb friction, gear backlash, dimension tolerance, link rigidity etc. influence motor behaviour and can at worse make the system unstable. Because we want the robot to act precise and stable the motors have to be controlled using actual position measurements.

Motor model

We use the motor model developed in task 2 to design our controller. In a first step all non-linear influences should be considered as external disturbances. Because of the mentioned variety of effects a significant amount for the disturbance has to be taken in account for implementing a robust controller. For simplified tuning of the controllers first the feedback loops for flexibility and induced voltage are neglected and disturbances for voltage and torque are included as shown in Figure 4.1. The previously mentioned influences on the motor behaviour are mostly covered with the torque disturbance.

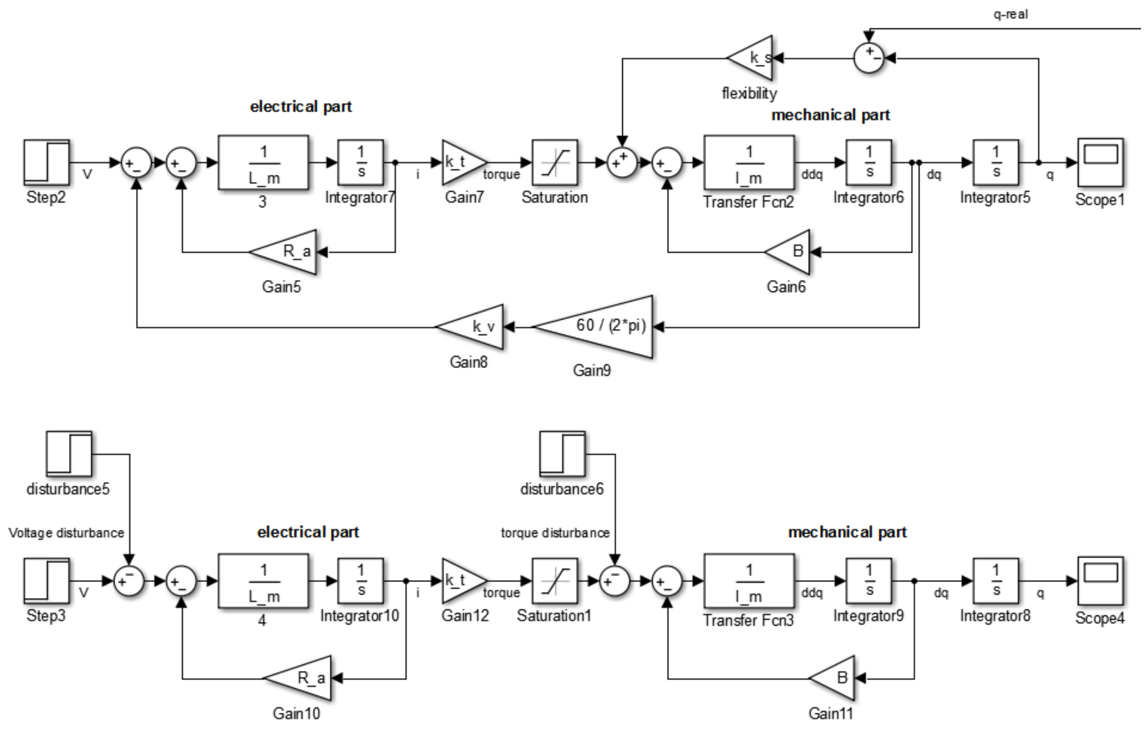


Figure 4.1: Motor model: complete and considering disturbances

We can simplify the resulting motor model by modelling the two feedback loops in the electrical and mechanical part with PT1 elements. For this purpose we use the general rule for mathematical transformation of control circles in Equation 4.1.

$$G(s) = \frac{G_f(s)}{1 + G_f(s) * G_r(s)} \tag{4.1}$$

with:
$G(s)$ – closed loop transfer function
$G_f(s)$ – forward path transfer function
$G_r(s)$ – return path transfer function

Equation 4.2 to Equation 4.5 show the transformation for the electrical part concluding to an general PT1 element with time constant T_e and factor k_e, for the mechanical part Equation 4.6 to Equation 4.9 shows the transformation. The new simplified motor model can be seen in Figure 4.2

$$G_f(s) = \frac{1}{L_m} * \frac{1}{s} \tag{4.2}$$
$$G_b(s) = R_a \tag{4.3}$$
$$G(s) = \frac{\frac{1}{R_a}}{\frac{L_m}{R_a} * s + 1} \tag{4.4}$$
$$G(s) = \frac{k_e}{T_e * s + 1} \tag{4.5}$$

$$G_f(s) = \frac{1}{I_m} * \frac{1}{s} \tag{4.6}$$
$$G_b(s) = k_v \tag{4.7}$$
$$G(s) = \frac{1/k_v}{\frac{I_m}{k_v} * s + 1} \tag{4.8}$$
$$G(s) = \frac{k_m}{T_m * s + 1} \tag{4.9}$$

with:
$k_e = \frac{1}{R_a}$, $T_e = \frac{L_m}{R_a}$, $k_m = \frac{1}{k_v}$, $T_m = \frac{I_m}{k_v}$

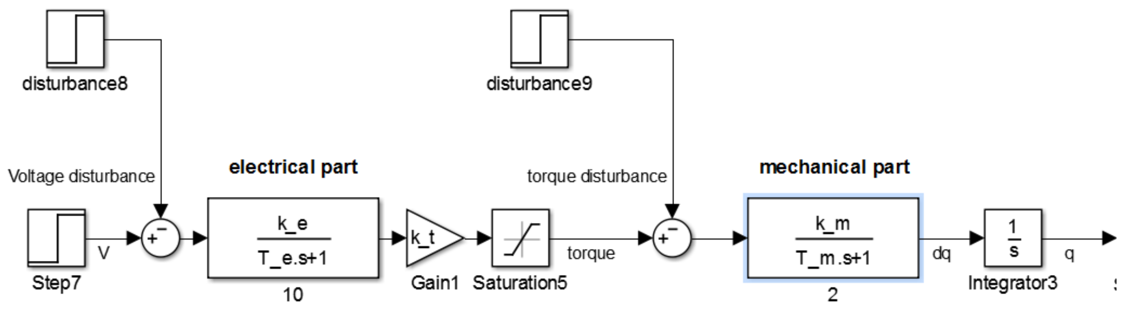


Figure 4.2: Motor model with PT1 elements

Position sensors

An encoder is attached to each motor in the ERB power balls of the robot. These modules of type "HEDS 9140" (see Figure 4.3) provide the position of the joint with a resolution of 500 counts per revolution. When the controllers are implemented on the robot these actual motor angle sensing provides feedback for our controller loop which can continuously be compared to our desired angle. The encoder to motor shaft ratio of 3:1 is calculated internally and so not considered in our model.

Figure 4.3: HEDS 9140 Encoder

General Control Structure

We want to build up two controllers for the electrical and mechanical part of the plant. The robot already includes a fast current controller attached to each motor. For our simulation we want to rebuild this very fast controller which can then be neglected for the further mechanical control structure.
The main control focus is on the position control of the robot using the real angle feedback from the encoders. We introduce a cascaded control structure with position and velocity feedback. The advantage of this two feedback loops is to eliminate disturbances as fast as possible, means where they occur, and so makes the overall controller more robust and precise. The feedback of position and velocity after transforming leads to a PID controller with position feedback only and so can be implemented with our robot sensors. Figure 4.4 shows the overall control-structure with the inner feedback loop for current control and both outer ones for velocity and position. We will deal with both control tunings for the electrical and mechanical part separated and integrate them later.

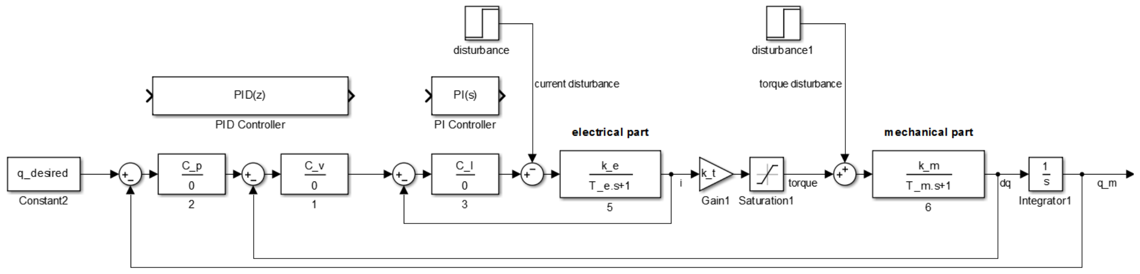


Figure 4.4: Complete control structure

Controller types and (dis-)advantages

The control behaviour of our robot should be robust, fast and stable. Therefore we first take a look at the main controller types to choose the best ones for our goals and merge the advantages. The main basic controller types are Proportional, Integral and Derivative, Table 4.1 shows their advantages and disadvantages.

Table 4.1: Controller types

Symbol	Controller type	Advantage	Disadvantage
P	Proportional	fast	maximum overshoot, steady state error
I	Integral	eleminates steady state error	high maximum overshoot, slow
D	Derivative	smaller maximum overshoot	steady state error

Current Control

The current control should be especially fast, because this control loop in between the actual position loop should not slow down the whole moving robot. Therefore we will use an proportional element with high gain which also can reduce the disturbance. For also eliminating a steady state error we need an additional integral part and so we will implement a PI-controller which has the general form of Equation 4.12.

$$C_c = K_{PI} * \frac{T_{PI} * s + 1}{T_{PI} * s} \tag{4.10}$$

Setting $T_{PI} = T_e$ we can cancel out the pole of the plant and so make the system stable. With the rule of Equation 4.1 the control scheme in Figure 4.15 leads to the closed loop transfer function in Equation 4.11, which shows PT1 behaviour.

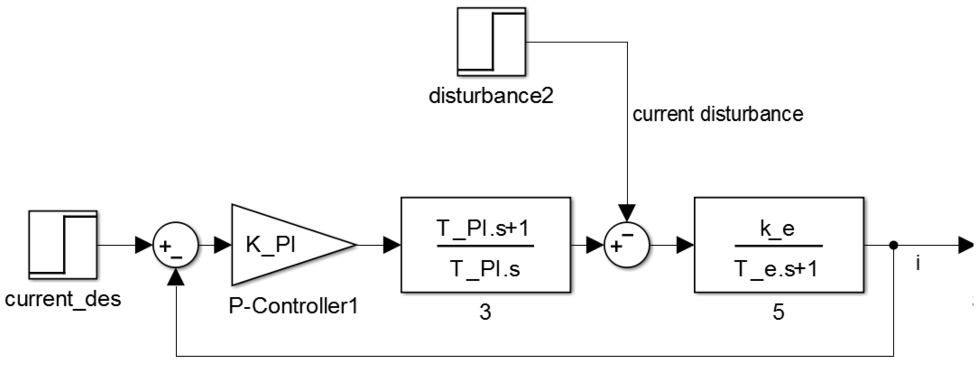


Figure 4.5: Current Control

$$G_c = \frac{1}{\frac{T_{PI}}{K_{PI} * k_e} * s + 1} \tag{4.11}$$

The root-locus plot, Figure 4.6, illustrates a stable behaviour as we only have a pole on the real axis. The position of the pole and so the time constant can now be adjusted choosing a K_{PI} gain. To get no influence on the position controller we want to let the system be 10 times faster than the outer mechanical control. Using the time constant to be $T_{current} = \frac{T_e}{K_v * k_e}$ $(T_{PI} = T_e)$ with $T_{current} = T_{position}/10$ we pre-set $K_{PI} = 30$ and so $T_{current} = 0.006$ and if needed adjust later using these constraints.

The performed tuning leads to the current controller:

$$C_c = 30 * \frac{T_e * s + 1}{T_e * s} \tag{4.12}$$

A look on the step response in Figure 4.7 emphasizes a quite fast and stable control. Due to physical motor constraints the current is limited internally.

Position and Velocity Control

For controlling the position of the motors we implement the cascaded control of Figure 4.8. Setting up the control types we start at the inner loop using the velocity feedback. It is reasonable to eliminate the pole of the PT1 behaviour of the motor model. We will need an integral and proportional part to exploit advantages of the control types especially reducing the steady state error and reaching fast control, respectively. With

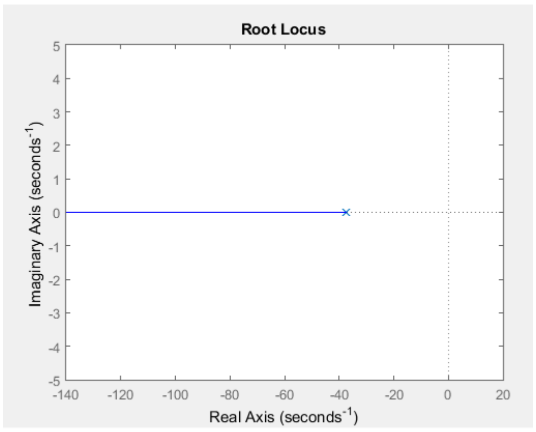


Figure 4.6: root locus plot of current control

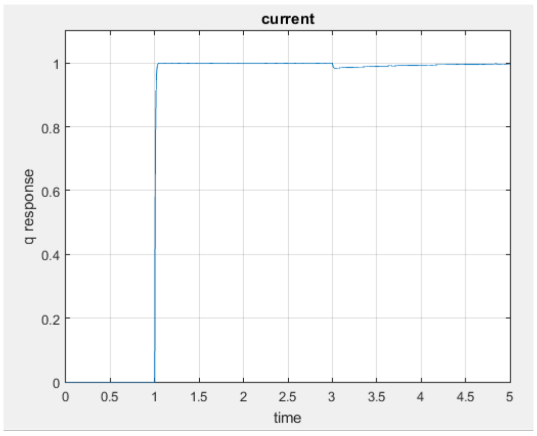


Figure 4.7: step response current control (disturbance at t=3)

these considerations we get the controller structure of Equation 4.13 for velocity control. For position feedback no additional disturbance is visible and so a fast P-controller as C_p is sufficient for position control.

$$C_v = K_v * \frac{T_v * s + 1}{s} \tag{4.13}$$

$$C_p = K_p \tag{4.14}$$

with: $T_v = T_m$

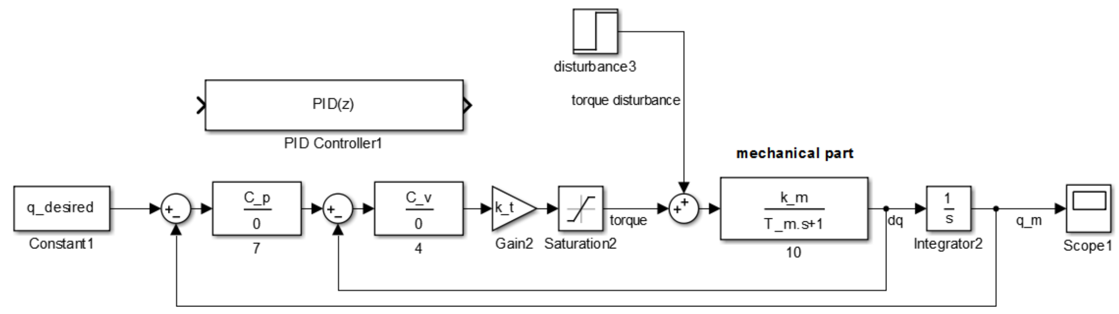


Figure 4.8: cascaded position and velocity control

After cancelling out the pole-zero compensation we get G_i from Equation 4.15 as the closed loop transfer function of the inner velocity loop. With this first step we can compute the whole closed loop transfer function G_o shown in Equation 4.16

$$G_i = \frac{1}{\frac{1}{k_m * K_v} * s + 1} \tag{4.15}$$

$$G_o = \frac{1}{\frac{1}{k_m * K_p * K_v} * s^2 + \frac{1}{K_p} * s + 1} \tag{4.16}$$

It shows up to be a second order system which can be described with the general formula [BS09]

$$G_o = \frac{1}{\frac{1}{\omega_n^2} * s^2 + \frac{2 * \zeta}{\omega_n} * s + 1} \tag{4.17}$$

with:
ω_n – natural frequency
ζ – damping ratio

Comparing our system with the general scheme we get the constraints for K_v and K_p

$$K_v = \frac{2 * \zeta * \omega_n}{k_m} \tag{4.18}$$

$$K_p = \frac{\omega_n^2}{K_v * k_m} \tag{4.19}$$

Another constraint to the controller is the desired settling time T_s, reaching 90% of the goal value with the step response. In a general way we can calculate it with [KJA11]

$$T_s = \frac{4}{\zeta * \omega_n} \tag{4.20}$$

The trajectory generation provides new positions in small time steps, here we assume maximal position changes of 5°, which therefore is the biggest step function that can apply. Comparing with the maximal motor speed of $72°/s$ (from motor data-sheet) which means 0.07s for 5°, we set our desired settling time to 1 second as the motor could start and end with zero velocity. For less oscillation in our system we choose our damping ratio $\zeta = 0.9$. Using transformed Equation 4.20 we calculate $\omega_n = 4.44 rad/s$ Plugging in these values in Equation 4.18 and Equation 4.19 we get

$$K_v = 6.4, K_p = 3.16 \tag{4.21}$$

We check our calculation with drawing the root locus function of the system. Figure 4.9 shows a complex pole pair. This means we will get a slightly oscillating system behaviour. Nevertheless the poles are quite close to the real axis and with choosing small proportional parts the system will be well damped. The resulting small oscillation therefore is accepted, but to make sure we also test the step response function.

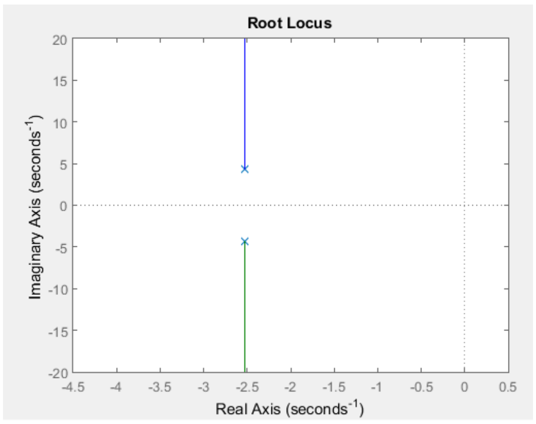


Figure 4.9: root locus plot for cascaded control

In Figure 4.10 we see a adequate fast response with an overshoot of less than 2% and no visible oscillation. Also the torque disturbance at time $t = 3$ is controlled fast and no steady state error stays.

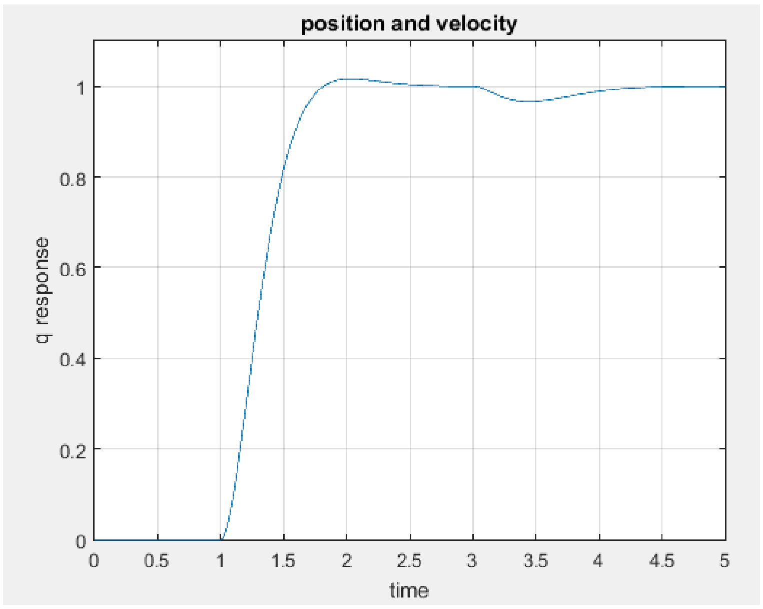


Figure 4.10: step response cascaded control (torque disturbance at t = 3s)

Equivalence and Integration

Our developed cascaded controller uses velocity and position feedback to control the system. As mentioned in the beginning the robot has only position sensors on each joint. Nevertheless the acceleration can be computed internally and so the overall cascaded controller for the mechanical part can be simplified by some transformation steps. Figure 4.11 shows a PID controller with only position feedback from the real robot and is equivalent to the cascaded structure we developed. We can use a prepared PID-block from the simulink library and set the parameters for the proportional, integral and derivative part of the controller with the following equivalence formulas [**robotics**]:

$$P = k_p * k_v * T_v + k_v \tag{4.22}$$

$$I = k_p * k_v \tag{4.23}$$

$$D = k_v * T_v \tag{4.24}$$

This mathematical transformation assumes the derivative part with a transfer function of $D * s$ only. This is theoretically correct, but practically not implementable. In reality a derivative is modelled by $D * \frac{s}{\frac{1}{N}s+1}$, including a filter coefficient N [KJA11]. With N we can filter out high frequencies and so we choose $N = 20$ for a reliable control. Nevertheless the derivative part reacts quite strong to a step response, as shown in

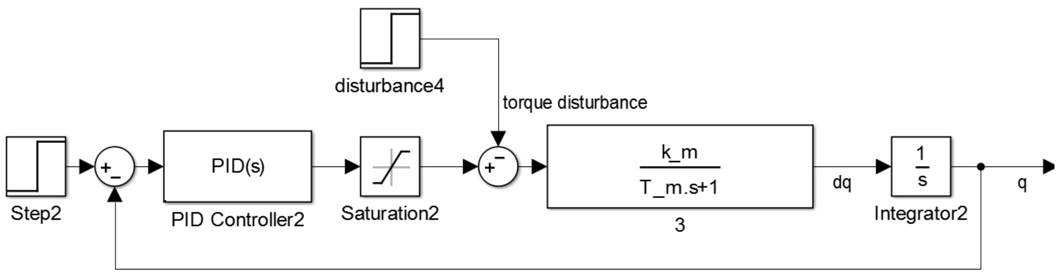


Figure 4.11: equivalent PID-controller

Figure 4.12. The system is overshooting by more than 20% which is different to the cascaded control and not acceptable. These results leads to the use of the cascaded control for the further work, assuming a velocity sensor on the robot. Nevertheless the "equivalent" PID controller could be used for a robot simulation or the real robot. In reality the trajectory generator provides a smooth interpolated trajectory, with no jumps as the step response, and so a derivative controller is useful.

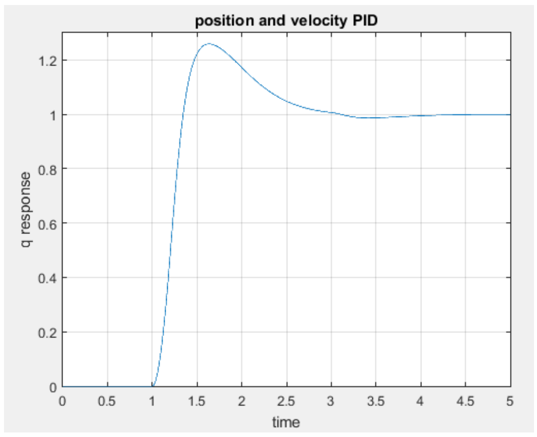


Figure 4.12: Step response PID-controller

As the next step we integrate the electrical and mechanical plant including their controllers (3 feedback loops at all for current, velocity and position) to the complete controlled system. Figure 4.13 shows the complete model, also including the feedback for induced voltage, which was neglected for tuning. The flexibility can not be considered here, because no real-q feedback is provided in this model. The step response in Figure 4.14 emphasizes a stable adequate fast control without oscillation. The overshooting is less than 1%, which is quite good. 90% of the desired value is reached in less than a second. A current disturbance of 0.5 at time $t = 3s$ is nearly not visible and the torque disturbance of 0.5 at $t = 7$ is well controlled, no steady state error stays.

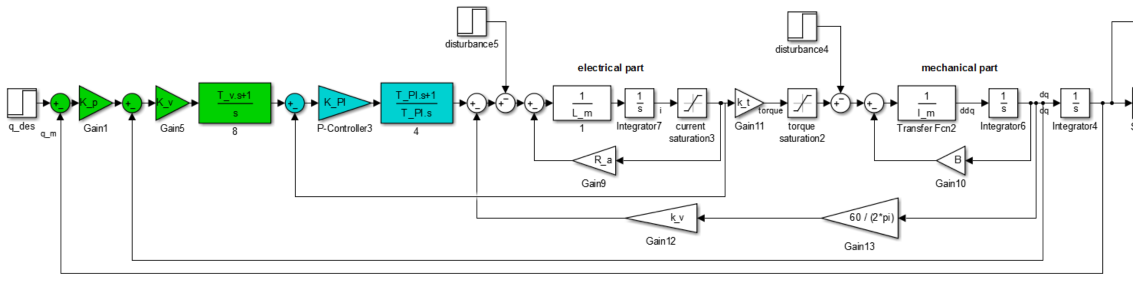


Figure 4.13: complete controlled model

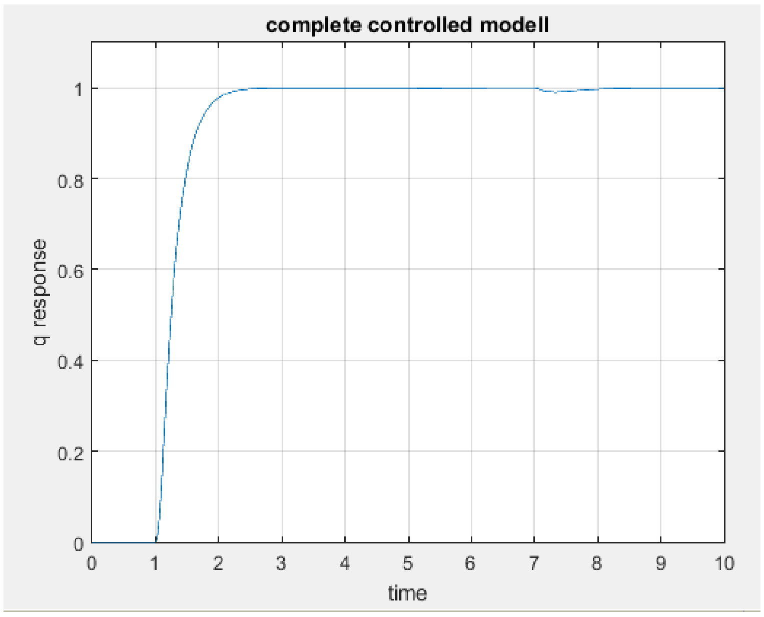


Figure 4.14: step response complete model (disturbance currence at t=5, torque t = 7)

Limitations due to the sampling time

For setting up and tuning our controllers we assumed the system to behave in continuous time. Nevertheless in reality the signals and calculation is discretized in small time samples. The continuous movement of the robot is converted into a digital signal by encoder sensing, which works as an A/D-converter. Control calculation then is done in fixed time samples and the output is converted with another D/A-converter to go back to the real-world system. We assume all these effects including the CAN-bus rate either including zero-order hold elements in the feedback loops or calculating the PID-controller in discrete time. This is done by transforming the function of s in the z domain. Internally a fixed step size numerical approximation of the system is executed [Bem11].

As the current control is done internally in the robot and we assume it to be really fast, we neglect the discretization for the current control part.

We choose a sampling of 1 kHz or $T_{sample} = 0.001s$. As we tuned our controller with a quite fast ω_n this sampling does not have a great impact to the robust control, except of discrete signals processing. Nevertheless with lower sampling rates one should be careful to not violate the Nyquist-Shannon sampling theorem and so cannot follow the robots continuous time behaviour with the controller.

As a test we have a look at the step response of our adjusted system. As there can be no difference be seen, except of a discrete sampled signal, we do not have to show it here.

For using the robot simulation as robot sensor feedback also sample and hold elements with T_{sample} should be inserted everywhere where discretization can be expected.

4.2 Improvements with model-based controller actions

For the first controller implementation many torque influencing parameters were not really considered, but assumed as disturbances. Although the controller can handle the disturbances every disturbance decreases the controllers performance. Since we know various parameters we now try to integrate them in the controller to improve accuracy and robustness.

Gravity compensation

The most obvious influence to all the links is gravity. It depends on the static gravity acceleration of $9.81m/s^2$ in ground direction and is affected to every body on earth. Since the robot is moving it changes over time differently for every joint. As sensor feedback we get a q vector containing all actual link angles of the robot and so gravity torque can be computed continuously for all joints. Again we use the Newton-Euler recursive function (N_rec) from task one. By passing the parameters q with all joint angles, assuming a static system in small time steps so that $\dot{q}$ and $\ddot{q}$ equal zero, and the gravity vector to the function, we get torques to compensate depending on the actual robot position (see Figure 4.16)

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \overbrace{\mathbf{C}(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{v}(\dot{\mathbf{q}}) + \mathbf{g}(\mathbf{q})}^{\mathbf{n}(\mathbf{q},\dot{\mathbf{q}})} = \mathbf{u},$$

Figure 4.15: Equation of Motion [Giu15]

$$\mathbf{g}(\mathbf{q}) = NE_{rec}(\mathbf{q}, \mathbf{0}, \mathbf{0}, \mathbf{g})$$

Figure 4.16: Function Call of NE_rec to get the gravity vector [Giu15]

We embed the calculation in a Matlab embedded function block for Simulink and so can use the q signal to compute the torque resulting from gravity. We simply add the computed gravity value for the link to the torque input of the motor, because this effort has always to be applied. For visualization in Figure 4.17 the gravity block is included in a single motor model. Of course all q positions have to be provided for the embedded function and so a torque vector u for all joints is computed.

It is not only possible to determine gravity, but in the same way you can verify friction and centripetal acceleration. An extended feedback here would be to calculate the complete n vector as it is done in task 1 again in an embedded function block. In this case the matlab function block for calculating the gravity is simply replaced by one that calculates the n-vector.

Feedforward control

If a trajectory with high speed should be followed, only including the desired position is not sufficient. In this case we can also provide velocity and acceleration from the trajectory generator to make the control faster. This procedure is called feedforward compensation and also implemented in Figure 4.17. As both values are fed in the specific control loop of the cascaded control, we here show only the mathematically transformed structure in the PID control loop. Both $\dot{q}$ and $\ddot{q}$ are pre-calculated with the motor constants k_m and T_m and can then be fed in the control loop. The feedforward compensation leads to a smaller tracking error, but of course works better, the more precise the robot model is. As it is a feed in with no additional control loop there will no bad influence to the stability of the system.

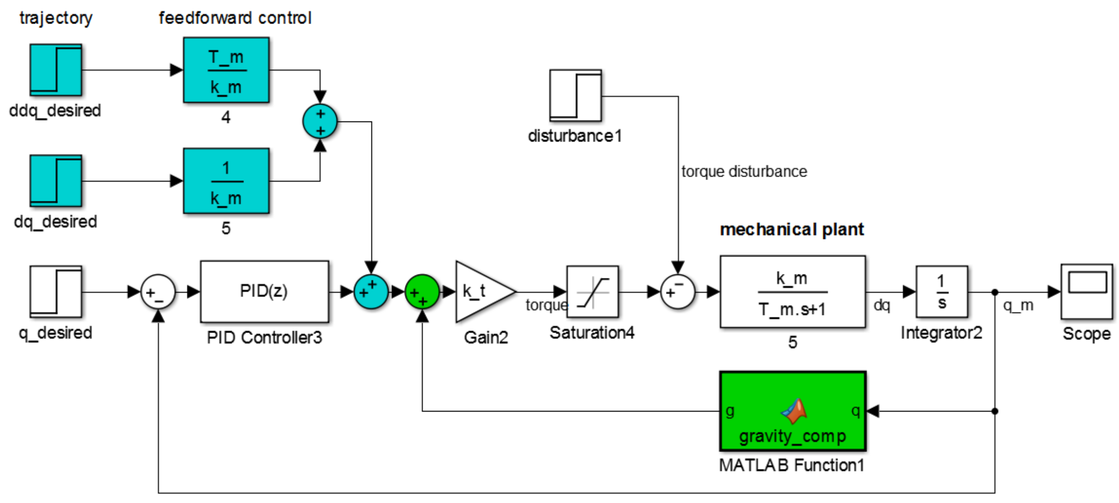


Figure 4.17: model-based control: feedforward control (cyan) and gravity compensation (green)

5 Modularizing Newton-Euler method

Responsible: *Benedikt Feldotto*

Idea

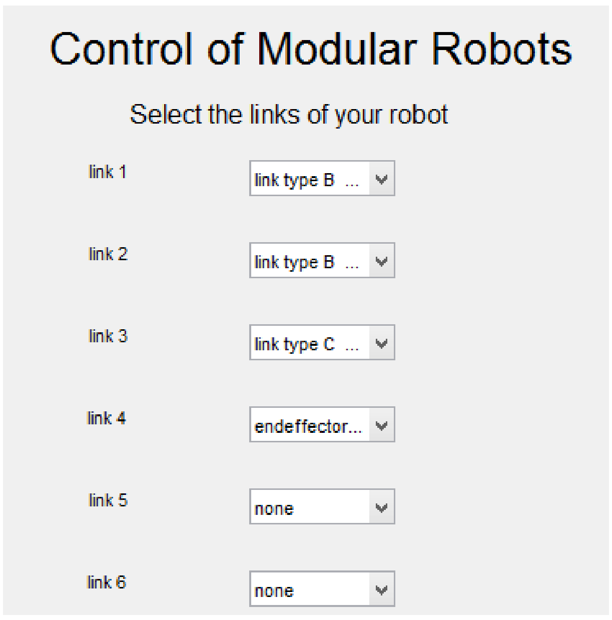
The Newton-Euler algorithm is an important part for the robot control. It is used for the simulation of the robot but also could help improve the controller behaviour for example with using gravity compensation. In task 1 we considered a fixed robot assembly, nevertheless as a global goal we want to deal with modular robots and so it is reasonable to also modularize the Newton-Euler algorithm. With this chapter we want to introduce a possible approach for a flexible robot dynamics calculation.
The Robot user does not want to care about any assignments especially does not want to build or rebuild Simulink models. Therefore a control panel as the one in Figure 5.1 could be used where the user just chooses the link types he attached to the robot in the right order from a drop-down menu and finally chooses the end-effector.

Implementation

The Newton-Euler method uses a forward and backward phase to calculate the dynamics. In both phases each iteration step is directly associated to one link. We can copy this structure to a simulink model and get a subsystem for each link module. Using input ports as shown in Figure 5.2 an inner matlab function block computes the output values. Instead of the whole forward and backward phase only one iteration step of each phase is implemented.

Realising the whole iteration loops we copy equal subsystems and connect them forwards and backwards regarding to the forward and backward loop of the Newton-Euler algorithm (Figure 5.3). Initialisation blocks with zero vectors ensure the first variable assignment in later calculation steps. Additionally we will need a module for the base, only providing the initial velocities for the Newton-Euler algorithm, and a module for the end-effector containing applied forces for initializing the backward loop.

We determine a maximum number of modules for the robot (which makes sense in case of physical constraints) and so for the number of Simulink Subsystems. Neverthe-


Control of Modular Robots

Select the links of your robot

link 1	link type B ...
link 2	link type B ...
link 3	link type C ...
link 4	endeffector...
link 5	none
link 6	none

Figure 5.1: control panel for user-friendly control [Giu15]

less in case of using less modules than prepared subsystem a logic ensures a correct signal routing. This contains enabling the used Subsystems for the used number of modules but also the assignment of the endeffector forces to the right module number.

In the model itself signal routing also ensures to connect pre-saved data for the different module types such as mass, inertia matrix and transformation to be provided to the right calculation subsystem as there is an input reserved for links characteristics.

A first programmatic sketch for this modular implementation is provided as source code with this work.

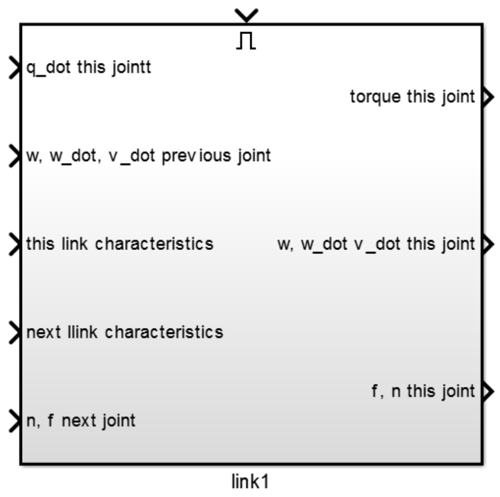

q_dot this jointt
torque this joint
w, w_dot, v_dot previous joint
this link characteristics
w, w_dot v_dot this joint
next llink characteristics
f, n this joint
n, f next joint
link1

Figure 5.2: subsystem for Newton-Euler iteration step

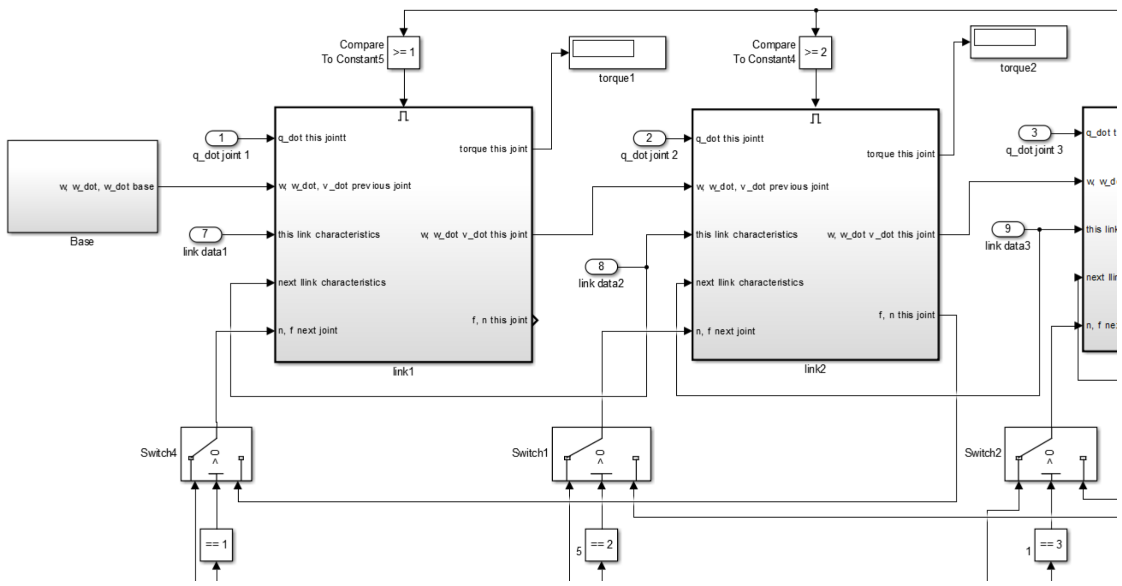


Figure 5.3: Newton-Euler: forward and backward signal routing

6 Conclusion

Kinematic control conclusion

In the first task, i.e. the kinematic modelling, the forward kinematics algorithm was implemented and tested. In addition, an iterative method to solve the inverse kinematics problem. However, there was a problem when iterating to find the joint angles for a desired pose. Therefore, the orientation part was neglected and the iteration was implemented to find the required joint angles to reach a desired end-effector position. Furthermore, a method to design joint space trajectories was implemented using a third order polynomial. As a next next step, a solution forthe inverse kinematics problem may be found by using Unit Quaternion so that the pose of the end-effector can be worked on as a whole. Automatic models shall be implemented to find the DH-tables. Moreover, using the Schunk robotic arm, kinematic calibrations can be done to better estimate the DH parameters.

In the dynamics tasks we could build a simulator for the robot. This simulator uses kinematic and dynamic parameters of the robot to calculate the actual joint position from appplied torques. Using a motor model of the actuators control loops for the electrical and mechanical parts were build and tuned. In conclusion desired positions can be reached fast without oscillation and small overshooting. The decentralized controllers are robust for a various number of attached joints as disturbances can be well controlled. In next steps the motor models and robot simulation should be improved with measurements on the real robot. With more precise models the control behaviour can be adjusted especially for the model-based control algorithms. Finally the controllers can be fine tuned on the real robot. With look on the modular robots we suggested an approach for modularizing the Newton-Euler algorithm as a central computation for simulation and control. Work should be spent on automating the simulation, so that only a control panel should work as a user interface and so easy modelling can be done.

In the last task we developed an kinematic control algorithm which delivers, based on its input of the desired position of the end effector, a movement of each joint of the robot. This can be used to control the robot in task B. While implementing the algorithm we found out that the inverse or pseudo inverse of the Jacobian matrix, which is used for the described conversion, is really cost-efficient and therefor not fitted for the algorithm, which has to operate in realtime. To avoid this, we used the transpose of the Jacobian matrix, which proved to be an efficient choice for our task.

Dynamics and Control conclusion

In the dynamics tasks we could build a simulator for the robot. This simulator uses kinematic and dynamic parameters of the robot to calculate the actual joint position from appplied torques. Using a motor model of the actuators control loops for the electrical and mechanical parts where build and tuned. In conclusion desired positions can be reached fast without oscillation and small overshooting. The decentralized controllers are robust for a various number of attached joints as disturbances can be well controlled. In next steps the motor models and robot simulation should be improved with measurements on the real robot. With more precise models the control behaviour can be adjusted especially for the model-based control algorithms. Finally the controllers can be fine tuned on the real robot. With look on the modular robots we suggested an approach for modularizing the Newton-Euler algorithm as a central computation for simulation and control. Work should be spend on automating the simulation, so that only a control panel should work as a user interface and so easy modelling can be done.

List of Figures

List of Tables

References

[Bem11] P. A. Bemporad. *Automatic Control (lecture University of Trento)*. 2011.

[BS09] L. V. G. O. B. Siciliano L. Sciavicco. *Robotics - Modelling, Planning and Control*. Springer-Verlag, 2009.

[Bur15] D. Burschka. *Robotics (lecture TUM)*. 2015.

[Cra05] J. J. Craig. *Introduction to robotics: Mechanics and control*. 3. ed., international ed. Pearson education international. Upper Saddle River, NJ: Pearson Prentice Hall, 2005. ISBN: 0-13-123629-6.

[G.P12] P. G. R. D. G.Prasad N.Sree Ramya. "Modelling and Simulation Analysis of the Brushless DC Motor by using MATLAB." In: (2012).

[Giu15] A. Giusti. *Recall of Dynamics*. 2015.

[HKSR] U. K. M. H. K. Samitha Ransara. "Modelling and Analysis of a Low Cost Brushless DC Motor Drive." In: ().

[KJA11] B. W. K. J. Aström. *Computer-controlled systems - Theory and Design*. Dover Publications Inc., 2011.

[M.N08] S. M. M.Nizam.Kamarudin. "Simulink Implementation of Digital Cascade Control DC Motor Model - A didactic approach." In: (2008).

[MWS06] M. V. Mark W. Spong Seth Hutchinson. *Robot Modeling and Control*. Wiley, 2006.

[Sch15] Schunk. *Powerball Lightweight Arm LWA 4P*. 2015.

[SHV06] M. W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot modeling and control*. Hoboken, NJ: Wiley, 2006. ISBN: 978-0-471-64990-8.

[SK08] B. Siciliano and O. Khatib, eds. *Springer Handbook of Robotics*. Berlin, Heidelberg: Springer Science+Business Media, 2008. ISBN: 978-3-540-23957-4. DOI: 10.1007/978-3-540-30301-5.

[SSVO10] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics: Modelling, planning and control*. Advanced textbooks in control and signal processing. London: Springer, 2010. ISBN: 978-1-84628-641-4.

[Cur14] Curtis Bradley. *Robotic ArmCalibration and Control*. 2014.

Glossary

Armature voltage (V)
Armature inductance (L)
Armature resistence (R)
Back EMF (V_b)
Back EMF constant (K_b)
Torque at link i (τ_i)
Induced current (i)
Generated torque (u)
Angular position of the joint (θ)
Angular velocity of joint $(\dot{\theta})$
Angular acceleration of joint $(\ddot{\theta})$
Rotation matrix from frame i+1 to frame i (R^i)
Translation into Z-direction of frame i+1 $(\grave{Z}_{i+1})$
Vector of positions of origins of frame i+1 to link i (P_{i+1})
Linear acceleration of link-frame origin i $(\dot{v}_i)$
Linear acceleration of the center of mass of link i $(\dot{v}_{c_i})$
Mass of link (m)
Force acting on the center of mass of link i (F_i)
Inertia Tensor (I)
Torque acting on the center of mass of link i (N_i)
Force exerted on link i by link i-1 (f_i)
Torque exerted on link i by link i-1 (n_i)
Inertia (J)
Torsional Stiffness (K_s)
Fricton constant (B)
Angular position of the rotor (q)
Torque Constant (K_m)
Angular velocity of joint (ω_m)
Angular velocity of conductor (ω_o)
Radius of joint (τ_m)
Radius of conductor (τ_o)
Gear ratio (K_τ)
Angular velocity of joint $(\dot{\theta_m})$
Angular velocity of conductor $(\dot{\theta}_o)$
Inertia of joint (J_m)

Inertia of conductor (J_o)