NTNU – Trondheim
Norwegian University of
Science and Technology

TTK4550
Specialization Project

Robot Manipulator Collision Handling in Unknown Environment without using External Sensors

Michael Remo Palmén Ragazzon

December 21, 2012

The Department of Engineering Cybernetics
Norwegian University of Science and Technology

Supervisor

Prof. Jan Tommy Gravdahl

Advisor

PhD Cand. Serge Gale

Abstract

A robot navigating in an unknown environment may collide into objects that have not been detected. We propose a scheme to detect collisions with such objects and navigate around them, so the robot can still reach its destination. In this project, a 6-degree-of-freedom (dof) robot manipulator has been considered. External sensors are assumed to be unavailable to keep costs at a minimum, although the proposed scheme could possibly be implemented as a supplement to other sensory information such as from a camera or tactile sensors.

For simulation purposes, the dynamics of a 6-dof manipulator is developed and an inverse dynamics controller is used to track a desired trajectory. An original method for collision detection based on the closed-loop dynamics of the controller is proposed. When a collision is detected, a signal will be sent to the collision avoidance system based on a Hopfield neural network, which is used to generate a new trajectory around all detected objects. This process is repeated until the destination is reached. The proposed solution has been tested in a simulation experiment.

Preface

At the start of this semester, the intended task was to do something within the topic of *robot learning*, such as artificial neural networks, reinforcement learning, and/or evolutionary robotics. The first part of the task was mainly a guided self-study on this very broad field. I had little prior knowledge within this topic, but it definitely sparked my interests having read science fiction and the proposed dystopian future resulting from research in such topics. As time went on, the focus shifted to robot manipulators, and specifically the Universal Robots UR5 available to us. We wanted a system which could react somewhat intelligently to the environment around it.

We were inspired to make a system which could detect and react to unforeseen collisions. Such a system would consist of several different components such as collision detection, collision avoidance, and robot dynamics. We didn't find a great deal of these subproblems that could employ concepts within robot learning, so the focus shifted a little away from this topic.

It was our intention to test the scheme on the UR5 manipulator, but sometimes time just doesn't cut it. Instead, a simulation experiment was done to show how well the different components and the overall system performed.

I would like to thank my supervisor Tommy Gravdahl and my advisor Serge Gale for their support. Tommy for giving me lots of advice along the way in our regular meetings, and Serge for giving me a great introduction to a field I had little knowledge about.

Contents

List of Figures

List of Tables

Nomenclature

$C(q,\dot{q})$	Coriolis and centripetal acceleration matrix
F	external force
$\hat{F}$	estimated external force
J	geometric Jacobian
$M(q)$	inertia matrix
$g(q)$	gravity vector
n	number of robot's degrees-of-freedom
q	joint angles
q_d	desired joint angles
$\tilde{q}$	joint angle errors
τ	input torques
τ_K	induced torque due to external force
$\triangleq$	equal by definition
$\lVert\cdot\rVert$	Euclidean norm
$\in$	symbol for ”is an element of”
$\subset$	symbol for ”is a subset of”
DH	Denavit–Hartenberg
dof	degree-of-freedom
EL	Euler-Lagrange method
iff	if and only if
NE	Newton-Euler method

Chapter 1

Introduction

Robots have traditionally been employed in closed environments separate from human beings, because they have been heavy, bulky, and dangerous. In recent years, a lot of development has been made to robots that are safe enough to interact with human beings. In the future, one might envision service robots co-existing with humans. Such robots will need to act intelligently on all information available to them to complete a given task. Just like human beings supplement their vision with other senses like touch and hearing, intelligent robots will need to do the same. We will propose a system that can help a robot navigate an unknown environment without requiring any external sensors.

Our work will be based on a 6-degree-of-freedom (dof) robot manipulator. The task will be to move the end-effector to some destination point in an environment with obstacles. The system will have to detect when a collision has happened without using external sensors, to keep costs at a minimum. We will propose a scheme to detect such collisions, register the collided object in a collision map, and move the manipulator around all detected objects until the destination is reached. It is envisioned that such a scheme can be used in combination with other sensory information in order to make the robot able to navigate more efficiently.

The overall task requires several different components to work together. All of these components are presented by themselves at first, and finally they are combined into a simulation experiment. We will start chapter 2 by presenting a basic introduction to artificial neural networks. Such a network is used in the collision avoidance scheme later on. In chapter 3 we will develop the dynamics of the Universal Robots UR5 manipulator. These dynamics will be used for two purposes, (1) for simulation of the UR5 manipulator, and (2) for use in the controller so the manipulator can be moved as desired. This inverse dynamics controller is presented in chapter 4. An original method for collision detection, which exploits the closed-loop dynamics of the controller, is presented in chapter 5. In chapter 6, a trajectory is generated around all detected objects, which uses a method based on neural networks. Finally in chapter 7, all components are implemented and combined in Matlab/Simulink to perform a simulation experiment. Results are presented and the efficiency of the different components is discussed. Conclusions and suggestions for future work are given in chapter 8.

1.1. Related Work

A lot of research focuses on detecting collisions and safely reacting to it, such as in (De Luca et al., 2006) and (Haddadin et al., 2008). These papers focus on impact with human beings and minimizing damage in the post-impact phase. We will assume that the mechanical design of and forces involved with the manipulator does not damage any being or object in the environment. Instead of going to a recovery or fault mode, we consider a collision just as new information for our robot, so it can learn about the environment and continue the original task of reaching the destination point.

In (Palejiya and Tanner, 2006) a hybrid velocity/force control is used in order to navigate an unknown environment. A force sensor is assumed available, and their robot will slide along an obstacle as soon as a collision is detected. No map of the environment is generated, and their method may end up with a very long travel path in certain environments.

In (Takakura et al., 1989) they provided a method for collision detection without sensors as well as a method of avoiding the obstacle. Their collision detection required either measurements of joint acceleration, or numerical differentiation of the joint velocities, which generates a very noisy signal. Their collision avoidance was very limited in the sense that it only moved a little to the side when a collision was detected.

We will take a different approach than these papers, by generating a map over the collision space, and follow the shortest free path in this space. The map will be updated every time a collision is detected.

1.2. Notation and Terminology

We will denote the number of degrees-of-freedom of a manipulator by n. For the UR5 manipulator $n = 6$. The following terminology will be used throughout the report:

Configuration space The configuration space defines every possible *configuration* a manipulator can take. A configuration specifies the location of every point on the manipulator. For a rigid manipulator with only revolute joints, a configuration is specified by the joint angles q, a vector in n-dimensional space.

Workspace The workspace of a manipulator is the physical space any configuration of the manipulator can reach, represented by 3-dimensional Cartesian coordinates. The axes and origin are placed equal to the inertial frame of the manipulator.

Collision space The collision space is either located in the workspace or configuration space, in our case in a 2-dimensional plane of the workspace. All collidable objects are specified in this space, and collision-free trajectories will be generated here before they are mapped to the workspace and/or configuration space.

Neuronal space The neuronal space is used in the collision avoidance scheme and consists of a finite amount *neurons* evenly distributed in the collision space. The neuronal space and collision space are sometimes used interchangeably.

Chapter 2

Neural Networks

This chapter will serve as an introduction to artificial neural networks. The purpose is both to provide insight into an important topic of robot learning, and specifically to provide background theory for a neural network which will be implemented later on. We introduce the artificial neural network by quoting the definition from (Haykin, 1998):

> A neural network is a massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:
>
> 1. Knowledge is acquired by the network from its environment through a learning process
> 2. Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge

The neural network is made up of many connected neurons. The model of a neuron usually consists of three basic elements,

- A set of *synapses*, which multiplies the input x_j by its own weight value
- An *adder* for summing the values from the synapses
- An *activation function* which takes the value from the adder and outputs a new value which is limited to some output range

See Figure 2.1 for an illustration of a neuron model. Many different activation functions can be used, such as a threshold function

$$\phi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0 \end{cases}$$

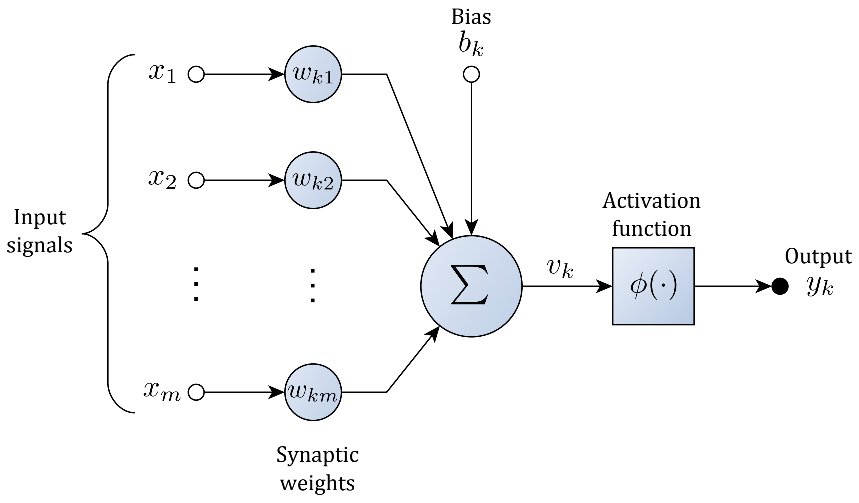


Figure 2.1: Neuron model. Image based on illustration in (Haykin, 1998).

or a saturated linear function

$$\phi(v) = \begin{cases} 1 & \text{if } v \geq \frac{1}{2} \\ v & \text{if } \frac{1}{2} > v > \text{-}\frac{1}{2} \\ 0 & \text{if } v < \text{-}\frac{1}{2} \end{cases}$$

or a Sigmoid function, characterized by its s-shape. An example of such a Sigmoid function is

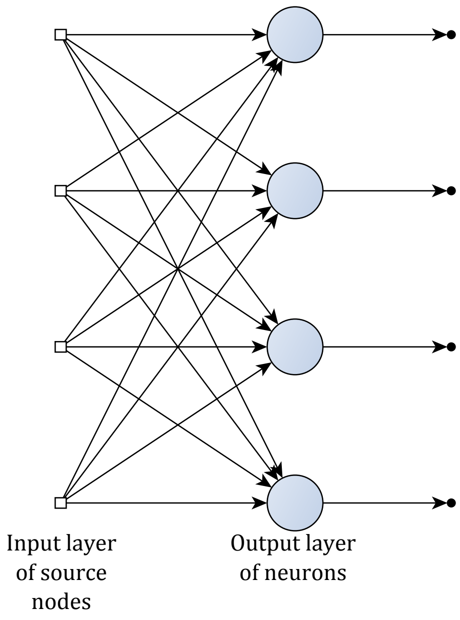
$$\phi(v) = \frac{1}{1 + \exp{(-av)}}$$

where a is the slope parameter.

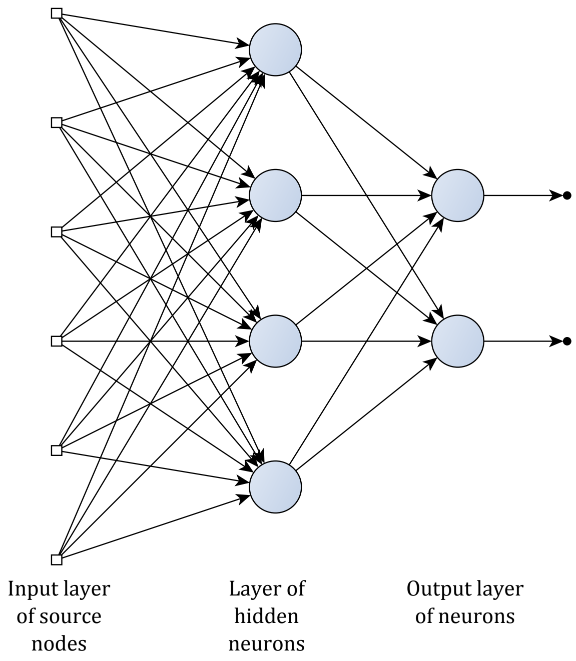
The neurons can be connected in several different ways to produce many different types of neural networks with different properties. The most fundamental network architectures are

- Single-layer feedforward networks, where each neuron is connected to every input, and the outputs from the neurons are the outputs of the network. Additionally, no neurons are connected to other neurons.
- Multilayer feedforward networks, which have one or more layers of hidden layer of neurons which are not directly measurable from the output.
- Recurrent networks, which is distinguishable from the feedforward networks in that it has feedback loops which connects the outputs from the neurons back into the other neurons.

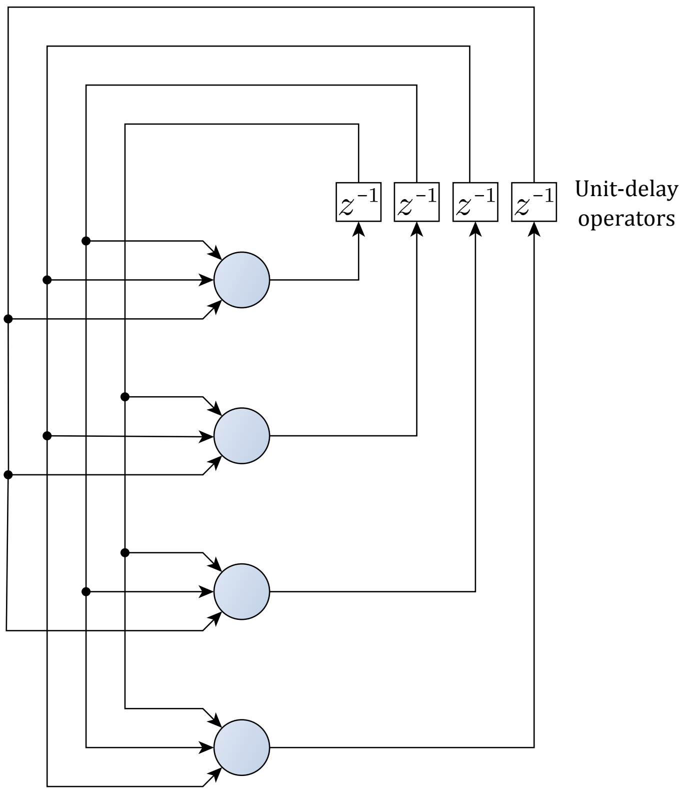
See Figure 2.2 for an illustration of the network architectures.



(a) Single-layer feedforward



(b) Multilayer feedforward



(c) Recurrent

Figure 2.2: The fundamental neural network architectures. Images based on illustrations in (Haykin, 1998).

2.1. Neural Networks and Learning

Neural networks are very suited for *learning*, and *improving* its performance. To define learning in the context of neural networks, we quote again from (Haykin, 1998):

> Learning is a process by which the free parameters of a neural network are adapted through a process of stimulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameter changes take place.

Many algorithms exist for the purpose of learning. One such method compares the output of the network to a desired response and updates the weights of the synapses to minimize the error. Another strategy, called Hebbian learning, compares the activity in two connected neurons and increases the synaptic weight between them if they are activated simultaneously. If they are activated asynchronously, the connection is weakened. According to (Haykin, 1998), there is evidence for Hebbian learning in the part of the human brain called the *hippocampus* which is important for our brain's ability to learn and memorize. This discovery provides motivation for use of the strategy in robot learning.

Neural networks can be used for several different learning tasks, including

- Pattern association, where a certain *memorized pattern* y_k is associated with a *key pattern* x_k. There are two phases involved. Firstly, the *storage phase* which trains the network to store some patterns y_k. Secondly, in the *recall phase*, a noisy or distorted version of the key pattern is presented to the network and used to retrieve the associated y_k.
- Pattern recognition, where a received pattern x_k is prescribed to one of a number of classes.
- Function approximation, where we are interested in approximating the non-linear function $f(x)$.

2.2. Hopfield Neural Network Model

So far, a very general introduction to neural networks has been given, mostly to give an overview of the topic of neural networks. In chapter 6, a type of neural network called the Hopfield network is implemented. Specifically, it is used to generate a collision-free path for the robot manipulator.

The Hopfield network is a form of a recurrent network, and looks exactly like in Figure 2.2 **(c)**. Here we can see a network with $N = 4$ neurons where every neuron is fed back into every other neuron through a unit-delay. The network is set-up by assigning weight values to the synapses and initial conditions on the output of each neuron. The outputs of the neurons define the system's *state*. By iterating the system through time, the state will converge to some stable point in the sense of Lyapunov.

The Lyapunov function of the Hopfield network can have many minima, so the network will have multiple locally stable states. Now, consider the case where we assign these minima exactly where we want them such that the stable states represent patterns to be memorized. This can be done by assigning the weight values in a certain way, and is called the storage or learning phase. After the storage phase, we can now retrieve an unknown signal. By setting the initial state of the network to this signal, and iterating, the network will converge to the state of one of the stored memories. Thus, even if the signal is noisy or distorted, it will still be able to retrieve the correct memory.

The application we just described is called content-addressable memory, a form of pattern association. A potential usage is in image recognition where each pixel is assigned to one neuron. The network could store images of certain known objects, and then a picture of an unidentified object could be applied to the network for the retrieval phase. By iterating, the system will converge to one of the known objects most similar to the unknown object.

The stability proof of the Hopfield neural network in the sense of Lyapunov will be provided in chapter 6 where we use it for the collision avoidance scheme.

Chapter 3

Dynamics of the UR5 Manipulator

Dynamics is the study of how forces and torques impact the motion of an object. In mathematical terms, this relationship is presented using the *equations of motion*. The goal of this chapter is to develop the complete dynamic model of the 6-dof Universal Robotics UR5 manipulator (Figure 3.1) in the form

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) \quad = \quad \tau \tag{3.1}$$

where $q = q(t) \in \mathbb{R}^n$ are the joint angles, $M(q) \in \mathbb{R}^{n\times n}$ is the inertia matrix, $C(q,\dot{q}) \in \mathbb{R}^{n\times n}$ is the Coriolis and centripetal acceleration matrix, $g(q) \in \mathbb{R}^n$ is the gravity vector, $\tau \in \mathbb{R}^n$ is the vector of input torques, and n is the number of dof. In general, such a model can be used for at least two different purposes

1. Direct dynamics, where we are interested in the resulting state (q, $\dot{q}$) of the robot given known input torques (τ)

2. Inverse dynamics, where we want to find the input torques (τ) to control the robot to certain states (q, $\dot{q}$) over time

Direct dynamics must be used for simulation purposes. Inverse dynamics is commonly used in controllers, such as the inverse dynamics controller which will be investigated further in chapter 4. We will do simulations, as well as make a controller based on inverse dynamics, so both of these purposes will be employed.

3.1. Background Theory

There are several methods to develop the dynamics of a robot manipulator. The most common methods are the *Euler-Lagrange method* and the *Newton-Euler method*. The Euler-Lagrange method is based on the mechanical energy of the manipulator, while the Newton-Euler method uses forces and torques between individual links to develop the dynamics. This section will briefly present and compare the two methods. The Newton-Euler method will be used to develop the dynamics of the UR5 manipulator.

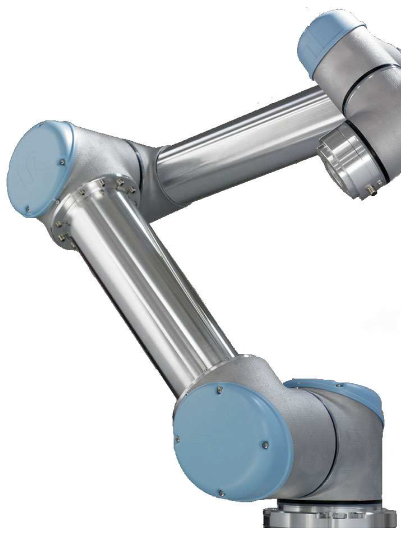


Figure 3.1: Universal Robots UR5 Manipulator. Image: (Uni)

3.1.1. Euler-Lagrange Method

The Euler-Lagrange method represents some of the earliest work in regards to dynamical models of robotics manipulators. It is based on the difference between the kinetic energy K and the potential energy P of the manipulator,

$$\mathcal{L} = K - P \tag{3.2}$$

called the Lagrangian. Once the Lagrangian is found, the Euler-Lagrange equation

$$\frac{d}{dt}\frac{\delta \mathcal{L}}{\delta \dot{q}_j} - \frac{\delta \mathcal{L}}{\delta q_j} \quad = \quad \tau_j, \quad j = 1, \ldots, n \tag{3.3}$$

can be used to find the equations of motion (Featherstone and Orin, 2008; Spong et al., 2006). For the Euler-Lagrange equation to hold, the system must be in a set of generalized coordinates, which means that the vector q must uniquely determine the configuration of the manipulator in the workspace. Additionally, the system must be *holonomic*, such that "*the number of degrees of freedom are equal to the number of coordinates needed to specify the configuration of the system*" (Holmberg and Khatib, 2000).

To develop the dynamics, the potential and kinetic energy must be found. The equations for doing so will now be presented, and is based on (Spong et al., 2006). First, the potential energy can be found using

$$P = \sum_{i=1}^{n} m_i g^T r_{ci} \tag{3.4}$$

where m_i is the mass of link i, g is the gravity vector in the inertial frame, and r_{ci} is the coordinates of the center of mass of link i.

The kinetic energy can be written in the form $K = \frac{1}{2}\dot{q}^T M(q)\dot{q}$. The inertia matrix $M(q)$

is found using

$$M(q) = \sum_{i=1}^{n} \left[m_i J_{v_{ci}}^T J_{v_{ci}} + J_{\omega_i}^T R_i I_i R_i^T J_{\omega_i} \right] \tag{3.5}$$

where $J_{v_{ci}} = J_{v_{ci}}(q)$ and $J_{\omega_i} = J_{\omega_i}(q)$ are the *Jacobian* matrices to the center of mass of link i, $R_i = R_i(q)$ are the rotation matrices from the inertial frame to frame i, and I_i is the inertia tensor of frame i. The Jacobians relate angular and linear velocities in the workspace to the joint velocities of the manipulator. See appendix A.1 for a proper definition of the Jacobian.

Let us define the Christoffel symbols,

$$c_{ijk} \triangleq \frac{1}{2} \left[\frac{\delta M_{kj}}{\delta q_i} + \frac{\delta M_{ki}}{\delta q_j} + \frac{\delta M_{ij}}{\delta q_k} \right] \tag{3.6}$$

where M_{ij} is the $(i, j)^{\text{th}}$ element of $M(q)$. The $C(q, \dot{q})$-matrix can now be found using

$$c_{kj} = \sum_{i=1}^{n} c_{ijk}(q)\dot{q}_i \tag{3.7}$$

where c_{kj} is the $(k, j)^{\text{th}}$ element of $C(q, \dot{q})$. Using these symbols, the matrix $\dot{M}(q) - 2C(q, \dot{q})$ can be shown to be *skew symmetric* as defined in appendix A.3. This property is necessary in several controller designs, and in fact will have consequences for the collision detection in chapter 5.

Finally, the gravity vector $g(q)$ is found using

$$g(q) = \left(\frac{\delta P}{\delta q} \right)^T \tag{3.8}$$

The matrices $M(q)$, $C(q, \dot{q})$ and the vector $g(q)$ can now be inserted into (3.1) to form the equations of motion for the robot manipulator.

3.1.2. Newton-Euler Method

The recursive Newton-Euler method is developed in (Spong et al., 2006) for manipulators with revolute joints. Only the results are presented here for brevity. Note that (Spong et al., 2006) contains several misleading notations and errors, for which (Høifødt, 2011) provides a comprehensive errata. All equations presented here have been corrected using the errata.

While the Euler-Lagrange method treats the entire manipulator as one system, the recursive Newton-Euler method treats each link separately. Using Newtonian mechanics, forces and torques are calculated on each link.

The dynamics of a manipulator using the Newton-Euler method is developed in two parts. The first part treats velocities and accelerations by making the same set of calculations for each link starting at the first. This is called the forward recursion. The second part starts at the last link and treats forces and torques and how these propagate to the first link. This second part is called the backward recursion.

Notation and Definitions

Following is a list of vectors and matrices used in the equations for the Newton-Euler method. All of the vectors are expressed in frame i.

Velocity and acceleration	
$a_{c,i}$	acceleration of the center of mass of link i
$a_{e,i}$	acceleration of the end of link i (i.e. at the origin of link $i+1$)
ω_i	angular velocity of frame i w.r.t. frame 0
α_i	angular acceleration of frame i w.r.t. frame 0

Forces and torques	
g_i	acceleration due to gravity
f_i	force exerted by link $i-1$ on link i
τ_i	torque exerted by link $i-1$ on link i
R_{i+1}^i	rotation matrix from frame $i+1$ to frame i, $R_i^{i+1} = \left(R_{i+1}^i\right)^T$

Physical features of the manipulator (constant for a given manipulator)	
m_i	mass of link i
I_i	inertia tensor about the center of mass of link i oriented with frame i
$r_{i-1,ci}$	vector from the origin of frame $i-1$ to the center of mass of link i
$r_{i,ci}$	vector from the origin of frame i to the center of mass of link i
$r_{i-1,i}$	vector from the origin of frame $i-1$ to the origin of frame i

Forward Recursion

Start with the initial conditions

$$\omega_0 = 0,\ \alpha_0 = 0,\ a_{c,0} = 0,\ a_{e,0} = 0$$

and solve the following equations in order of appearance

$$\omega_i = R^i_{i-1}\omega_{i-1} + b_i\dot{q}_i \tag{3.9}$$
$$\alpha_i = R^i_{i-1}\alpha_{i-1} + b_i\ddot{q}_i + \omega_i \times b_i\dot{q}_i \tag{3.10}$$
$$a_{e,i} = R^i_{i-1}a_{e,i-1} + \dot{\omega}_i \times r_{i-1,i} + \omega_i \times (\omega_i \times r_{i-1,i}) \tag{3.11}$$
$$a_{c,i} = R^i_{i-1}a_{e,i-1} + \dot{\omega} \times r_{i-1,ci} + \omega_i \times (\omega_i \times r_{i-1,ci}) \tag{3.12}$$
$$g_i = R^i_{i-1}g_{i-1} \tag{3.13}$$

for $i = 1$ to n.

Backward Recursion

Start with the terminal conditions

$$f_{n+1} = 0,\ \tau_{n+1} = 0$$

and solve the following equations

$$f_i = R^i_{i+1}f_{i+1} + m_i a_{c,i} - m_i g_i \tag{3.14}$$
$$\tau_i = R^i_{i+1}\tau_{i+1} - f_i \times r_{i-1,ci} + \left(R^i_{i+1}f_{i+1}\right) \times r_{i,ci} + I_i\alpha_i + \omega_i \times (I_i\omega_i) \tag{3.15}$$

for $i = n$ to 1.

These equations can be used directly for inverse dynamics in which q, $\dot{q}$, and $\ddot{q}$ is provided and the corresponding τ is found. However, for simulation purposes, the equations needs to be in a form suitable for forward dynamics, such as (3.1). To find the elements of $M(q)$, $C(q,\dot{q})$, and $g(q)$, first the forward and backward recursion is done symbolically using automated software. Then the equations can be collected and combined to the desired matrices (Egeland and Gravdahl, 2002, sec. 7.10). For the UR5 manipulator, this is done using the automated framework of (Høifødt, 2011) which is described in section 3.2.3.

3.1.3. Comparison of Euler-Lagrange and Newton-Euler Method

The Euler-Lagrange method (EL) and the Newton-Euler method (NE) are both based on fundamental physical laws, i.e. conservation of energy and Newton's laws of motion respectively. EL makes a few requirements to the system, such that the system is holonomic, and in a set of generalized coordinates. As long as these requirements are met, the dynamics using either EL or NE will provide equal response. However, the equations of motion themselves are usually more complicated computationally when developed using EL.

Some of the earliest results in robotics used EL to develop the dynamics. The major drawback to this method was the high computational cost which limited the use of real-time inverse dynamics on manipulators. It was shown by (Hollerbach, 1980) that the method has a $O\left(n^4\right)$ computational cost, in terms of multiplications and additions, while NE has a computational cost of $O(n)$. The same paper presents a modification to the traditional Euler-Lagrange method using a recursive structure like the Newton-Euler method, and showed that such a method could also gain a computational cost of $O(n)$. Two years later, (Silver, 1982) provides more insight into the source of the computational differences, which was shown to be due to the recursive structure of NE and the choice of rotational dynamics. Using optimal choices of both methods, it was shown that in fact there is no fundamental difference between the computational cost of EL and NE.

It should be noted that forward dynamics is in general more computationally expensive than inverse dynamics, since this requires the inversion of the $M(q)$-matrix in (3.1) such that the dynamics is in a form suitable for numerical integration of $\ddot{q}$ (Egeland and Gravdahl, 2002, sec. 7.10).

A notable difference between EL and NE, is the skew symmetry property of $\dot{M}(q)-2C(q,\dot{q})$ which is easily gained in EL by using the Christoffel symbols to find $C(q,\dot{q})$. This skew symmetry property is not true for the dynamics of NE. We will see that this prevents us from using certain methods for the collision detection in chapter 5. This property is also important for several controller designs in manipulators.

3.2. UR5 Manipulator Dynamics

A Universal Robots UR5 manipulator has been used in this project. It is a 6-dof manipulator as pictured in Figure 3.1. The dynamics of the UR5 manipulator is found using the Newton-Euler method as this method is simple even for large n with its recursive structure. This makes it very suited for automated frameworks. Additionally, the low computational cost is gained automatically, whereas for the Euler-Lagrange method a low computational cost requires more effort.

In order to proceed with the development of the dynamics, reference frames and parameters must be defined for each link of the manipulator. First, the standard DH-parameters of the manipulator will be developed, then the necessary vectors and parameters used in the Newton-Euler method is defined.

3.2.1. Denavit–Hartenberg Parameters

The commonly used DH-convention defines four parameters that describe how the reference frame of each link is attached to the robot manipulator. Starting with the inertial reference frame, one additional reference frame is assigned for every link of the manipulator. The four parameters θ_i, z_i, a_i, α_i defined for each link $i \in [1, n]$ transforms reference frame $i-1$ to i using the four basic transformations

$$A_i = \text{Rot}_{z,\theta_i}\text{Trans}_{z,d_i}\text{Trans}_{x,a_i}\text{Rot}_{x,\alpha_i}$$

where the matrices $\text{Rot}_{\cdot,\cdot}$ describes rotation and the matrices $\text{Trans}_{\cdot,\cdot}$ describes translation, all of which are properly defined in appendix A.2. Further details of assigning DH-parameters are found in (Spong et al., 2006).

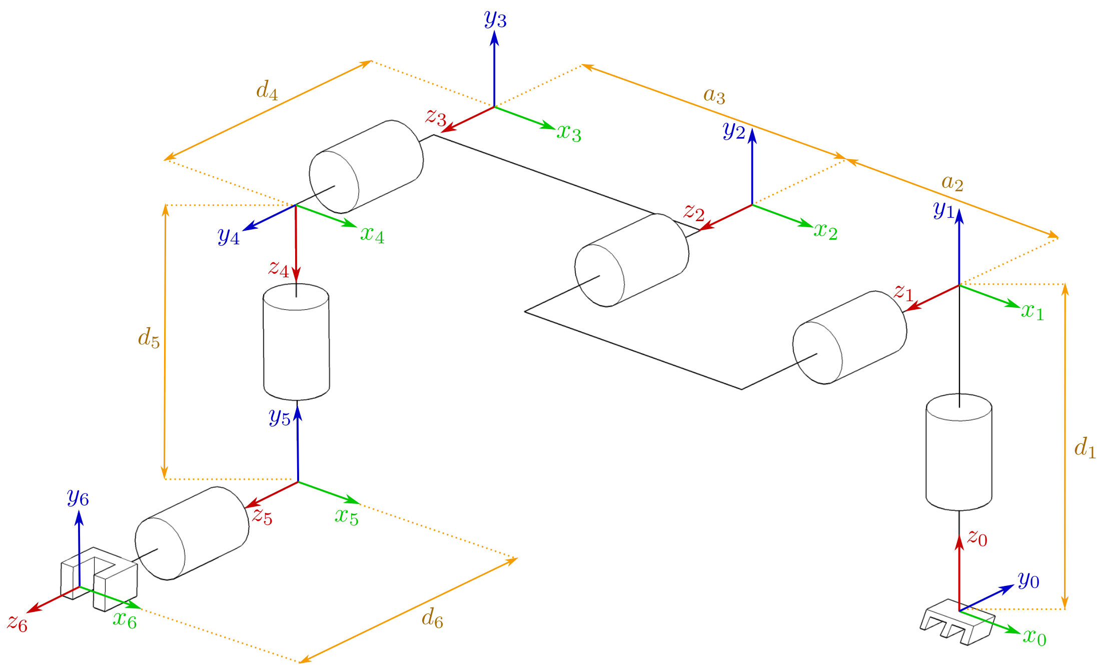


Figure 3.2: DH coordinate frame assignment of the UR5 manipulator. Link dimensions are not to scale.

The coordinate systems have been placed along the manipulator as illustrated in Figure 3.2. The DH-parameters for the manipulator is given in Table 3.1.

3.2.2. Newton-Euler Method Parameters

While the DH-parameters are used when developing the Euler-Lagrange equations of motion, the Newton-Euler method uses a slightly different set of parameters as shown in section 3.1.2. The constant vectors and matrices used in the model will be defined in this section.

The same coordinate frames as developed by the DH-convention will be used. The vectors $r_{i-1,i}$ which define the translation between coordinate frames can be found by studying Figure 3.2. Remember that each of the $r_{i-1,i}$ vectors are located in body frame i. The values are then found as

$$\begin{aligned}
r_{01} &= [\quad 0 \quad d_1 \quad 0 \quad] \\
r_{12} &= [\quad \text{-}a_2 \quad 0 \quad 0 \quad] \\
r_{23} &= [\quad \text{-}a_3 \quad 0 \quad 0 \quad] \\
r_{34} &= [\quad 0 \quad d_4 \quad 0 \quad] \\
r_{45} &= [\quad 0 \quad \text{-}d_5 \quad 0 \quad] \\
r_{56} &= [\quad 0 \quad 0 \quad d_6 \quad]
\end{aligned}$$

Table 3.1: DH-parameters for the UR5 manipulator.

Link	θ_i	d_i	a_i	α_i
1	θ_1^*	d_1	0	$\frac{\pi}{2}$
2	θ_2^*	0	-a_2	0
3	θ_3^*	0	-a_3	0
4	θ_4^*	d_4	0	$\frac{\pi}{2}$
5	θ_5^*	d_5	0	-$\frac{\pi}{2}$
6	θ_6^*	d_6	0	0

* joint variable

The rotation matrices between each frame is given by

$$R_{i-1}^{i} = R_{z,\theta_i} R_{x,\alpha_i}$$

where θ_i are the joint variables and α_i are configuration independent parameters given by the DH-parameters in Table 3.1. The rotation matrices $R_{x,\alpha}$ and $R_{z,\theta}$ are given in section A.2.

The DH-parameters, the link masses, and the center of mass vectors were provided by the manipulator manufacturer. The inertia tensors, were approximated for each link using a cylindrical inertia tensor

$$I_{cyl,x} = \begin{bmatrix} \frac{1}{2}mr^2 & 0 & 0 \\ 0 & \frac{1}{12}m\left(3r^2 + h^2\right) & 0 \\ 0 & 0 & \frac{1}{12}m\left(3r^2 + h^2\right) \end{bmatrix}$$

where m is the link mass, r is the radius of the cylinder, and h is the height of the cylinder along the x-axis as seen in Figure 3.3. Equivalent inertia tensors can be found along the y- and z-axes, denoted $I_{cyl,y}$ and $I_{cyl,z}$. The inertia tensors are defined along different axes for each link, to fit the physical manipulator as much as possible. The height and radius values for each link, h_i and r_i, were approximated from technical drawings of the manipulator. The values used for the UR5 manipulator is summarized in Table 3.2 and Table 3.3.

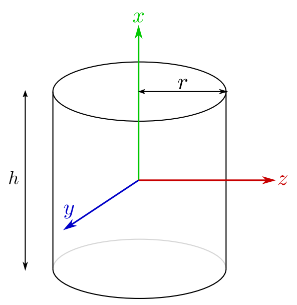


Figure 3.3: Cylinder used as an approximation of the inertia tensor of each link

Table 3.2: Summary of parameters for the UR5 manipulator (1/2)

Link i	d_i [m]	a_i [m]	m_i [kg]	r_i [m]	h_i [m]	I_i
1	0.0892	0	3.8054	0.045	0.13693	$I_{cyl,y}$
2	0	-0.425	8.4576	0.045	0.540	$I_{cyl,x}$
3	0	-0.39243	2.1456	0.0386	0.507	$I_{cyl,x}$
4	0.109	0	1.1884	0.0386	0.085	$I_{cyl,y}$
5	0.093	0	1.11884	0.03862	0.085	$I_{cyl,y}$
6	0.082	0	0.1632	0.0386	0.0395	$I_{cyl,z}$

Table 3.3: Summary of parameters for the UR5 manipulator (2/2)

Link i	$r_{i-1,i}$ [m]	$r_{i,ci}$ [m]
1	[0 0.0892 0]	[0 -0.01541 0.00151]
2	[-0.425 0 0]	[0.2125 0 0.1134]
3	[-0.39243 0 0]	[0.150 -0.00015 0.0154]
4	[0 0.1090 0]	[0 0.0015 0.0154]
5	[0 -0.093 0]	[0 0 -0.0128]
6	[0 0 0.082]	[0 0 0]

3.2.3. Equations of Motion using Automated Framework

An automated framework for running the backward and forward recursion equations from section 3.1.2 is given by (Høifødt, 2011). The framework is written in Maple and gives the equations of motion symbolically. The Maple file is shown in appendix B.1.

By running the Maple file, the matrices $M(q)$, $C(q,\dot{q})\dot{q}$, and $g(q)$ from (3.1) are found. The produced matrices for the UR5 manipulator can be found in appendix B.2. This concludes the development of the dynamics of the UR5 manipulator.

3.3. Simulation of UR5 Model Dynamics

A simulation was run in order to verify that the model developed for the UR5 manipulator in the previous sections works as expected.

3.3.1. Set-up and Controller

The model dynamics were simulated by setting the initial state of the manipulator to zero, such that

$$\begin{aligned} q(0) &= 0 \\ \dot{q}(0) &= 0 \end{aligned}$$

This means that the pose of the manipulator is equal to the pose in Figure 3.2, and if the model is correct, it is expected that the manipulator will start by falling down in some of the joints due to gravity.

A feedback signal from the $\dot{q}$ vector was used to damp the system so that it will end at a stable rest position. The controller can be described as

$$\tau = -K\dot{q}$$

where $K = K(t)$ is a scalar defined such that

$$K(t) \triangleq \begin{cases} 1 - 0.1t & \text{for } t \leq 9.8 \\ 0.02 & \text{for } t > 9.8 \end{cases}$$

This function is defined as such to give large damping while to system initially falls from the zero-position. This avoids some of the overshoot. As the system comes closer to rest, such large damping is not needed and the gain is reduced to make the system come to rest at a faster rate. The purpose of the variable damping is for the system to come to rest at a faster rate, and thus reduce the simulation time.

The model and controller was run using Simulink, and the block diagram is shown in Figure 3.4. The equations of motion for the UR5 manipulator were set-up in a Matlab S-function, which was based on the work of (Høifødt, 2011). The S-function takes q, $\dot{q}$, and τ as input parameters and outputs new $\dot{q}$ and $\ddot{q}$ which is then run through an integrator block. It essentially solves (3.1) for $\ddot{q}$, i.e. the forward dynamics problem.

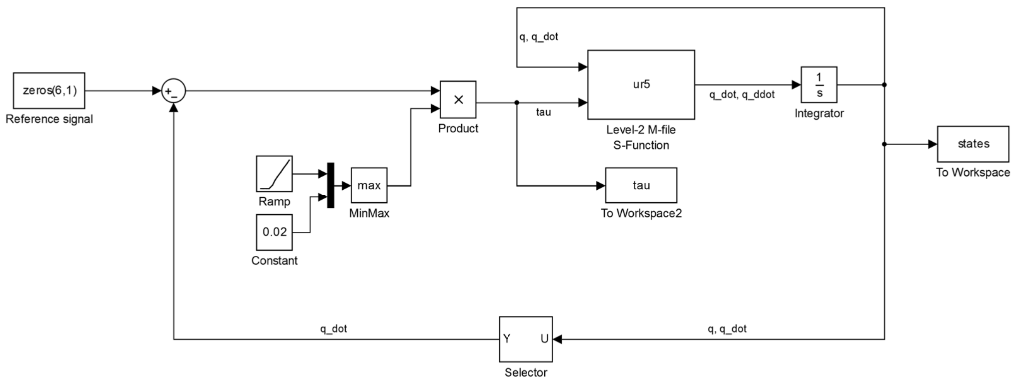


Figure 3.4: Simulink block diagram of UR5 manipulator dynamics with damping controller

3.3.2. Results

The results of the simulation are plotted in Figure 3.5. The plot shows the evolution of q and $\dot{q}$ over time. An illustration of the manipulator's pose at the end of the simulation is shown in Figure 3.6.

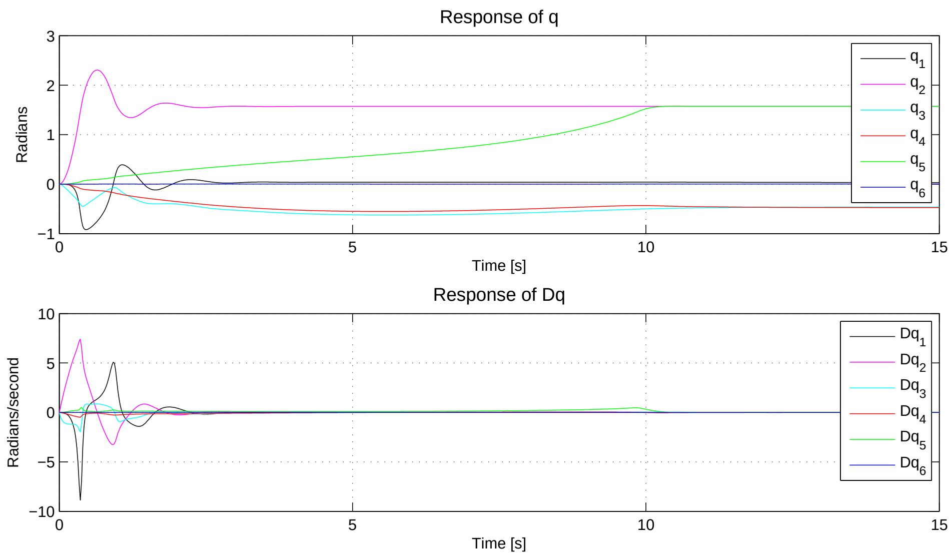


Figure 3.5: Simulation results of UR5 manipulator dynamics with damping controller

3.3.3. Observations

The simulation results shows that the system initially starts to fall due to gravity. The damping controller reduces $\dot{q}$ until the system comes to rest at around 11 seconds. The stationary end values of q define the rest pose of the manipulator. As one can see, q_2 ends at a stationary value of about $\pi/2$ [rad] $= 90°$. By comparing this result to the manipulator sketch in Figure 3.2, one can see that the second link falls almost straight down as expected. The fifth joint, q_5, ends at 90° which means that the end-effector is rotated such that it is pointing downward. This makes both joint 3 and 4 end up with an offset of -26.6° and -27.2° respectively to compensate for the weight of the end-effector.

Another interesting observation is that q_1 initially moves back and forth mainly as the second link falls down, overshoots, and falls back down again. So by swinging the second arm, one can expect the first joint to have a reverse motion, which behaves as one would expect by looking at Figure 3.2.

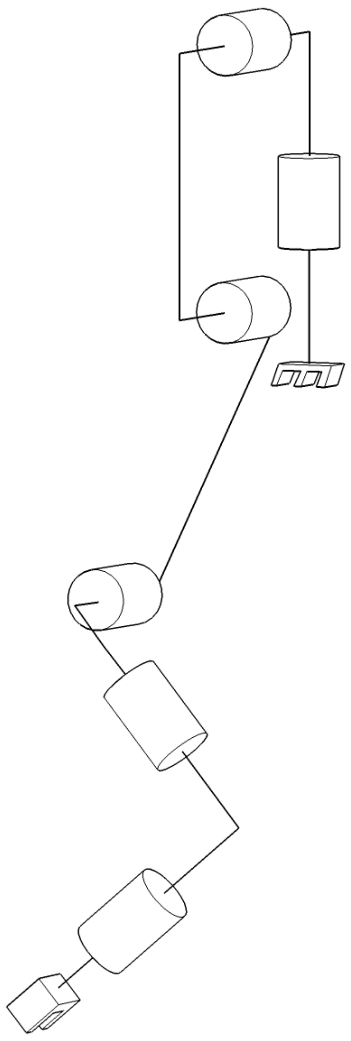


Figure 3.6: UR5 manipulator pose at the end of the simulation. Link dimensions are not to scale, joint angles are approximate.

Chapter 4

Inverse Dynamics Controller

In the previous chapter, the dynamics of the UR5 manipulator was developed. In order to control the manipulator to a desired trajectory, a controller is needed. In this chapter a controller based on inverse dynamics will be presented. This controller takes a smooth desired trajectory and returns the motor torque to follow it.

Consider the controller

$$\tau = M(q)a_q + C(q, \dot{q})\dot{q} + g(q) \tag{4.1}$$

where a_q is to be determined. By applying this controller to the dynamics equation (3.1) we get

$$\begin{aligned} M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) &= M(q)a_q + C(q, \dot{q})\dot{q} + g(q) \\ M(q)\ddot{q} &= M(q)a_q \end{aligned} \tag{4.2}$$

Since $M(q)$ is positive definite (Spong et al., 2006), it is invertible. Left-multiplying by it's inverse leads to

$$\ddot{q} = a_q \tag{4.3}$$

This is known as a double-integrator system, which essentially let us control the joint acceleration independently of each other. Let us set

$$a_q = \ddot{q}_d - K_0\tilde{q} - K_1\dot{\tilde{q}} \tag{4.4}$$

where $\tilde{q} = q - q_d$, $\dot{\tilde{q}} = \dot{q} - \dot{q}_d$, and K_0, K_1 are diagonal matrices consisting of position and velocity gains respectively. Note that a_q is just an acceleration feedforward with a PD feedback control.

To decide the gain matrices K_0, K_1, consider (4.4) inserted into (4.3) and rearranged, which leads to the closed-loop dynamics

$$\ddot{\tilde{q}}(t) + K_1\dot{\tilde{q}}(t) + K_0\tilde{q}(t) = 0 \tag{4.5}$$

By choosing the gain matrices as

$$K_0 = \begin{bmatrix} \omega_1^2 & 0 & \dots & 0 \\ 0 & \omega_2^2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \omega_n^2 \end{bmatrix}, \quad K_1 = \begin{bmatrix} 2\omega_1 & 0 & \dots & 0 \\ 0 & 2\omega_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 2\omega_n \end{bmatrix} \tag{4.6}$$

we have a decoupled closed-loop system where every joint response is a critically damped linear second-order system with natural frequency ω_i. The vector of natural frequencies is denoted by $\omega = [\ \omega_1 \quad \omega_2 \quad \dots \quad \omega_n\ ]^T$.

Now that the gain matrices have been defined, we can find a_q by providing a reference trajectory ($q_d(t)$, $\dot{q}_d(t)$, $\ddot{q}_d(t)$) to (4.4). And by using the dynamics of the manipulator as found in the previous chapter, we can now find the input torque τ using (4.1).

Chapter 5

Collision Detection

In this chapter, a method will be developed for detecting that our robot manipulator has collided into an object. By sensing the collision, appropriate response can then be taken. This can provide additional safety in the case where the manipulator hits an object that is not known to the robot. In our case, the goal of the manipulator is to reach a certain destination despite there being objects in the way. The collision point will be saved and then a signal will be provided to the collision avoidance method shown in the next chapter. This signal indicates that an object is in the way, and a new trajectory will be generated around the collision point.

When a collision occurs on the manipulator, forces are introduced at the contact point. Thus, detecting a collision is comparable to detecting contact forces. Several methods exist to detect such contact forces. The most obvious is to use force sensors. The downside to using force sensors is the added cost to the system. Additionally, only collisions occurring on the sensor itself will be detectable. Torque sensors placed on every joint may be used to estimate externally applied force. Such an approach will be able to detect a force applied at any link, although still introduces additional cost.

Several methods exist without using sensors. Such an approach is taken in (De Luca et al., 2006) which uses the generalized momenta, $M(q)\dot{q}$, and takes advantage of known dynamics of the manipulator to estimates the force. This method is shown to approach the externally applied forces exponentially fast. The link in which the collision occurs can be detected. Additionally, the direction of the force as well as the torque from the externally produced force can be found.

Such an approach was considered in our work. We know the dynamics of the UR5 manipulator, but the downside to the method in (De Luca et al., 2006) is that it requires the $C(q,\dot{q})$-matrix in (3.1) to have the property such that $\dot{M}(q) - 2C(q,\dot{q})$ is skew symmetric. Such a result is easily obtainable using the Euler-Lagrange method to develop the dynamics by using e.g. the Christoffel symbols (Spong et al., 2006). But as discussed in section 3.1 this property does not hold in general using the Newton-Euler method. Thus, we had to abandon this method.

Several papers use a disturbance observer by comparing the estimated torque to the commanded torque, such as in (Takakura et al., 1989; Suita et al., 1995). These methods require either numerical differentiation of joint velocities or a sensor for measuring joint acceleration. Numerical differentiation results in a noisy signal. Furthermore, it is difficult to tune the thresholds in such schemes.

In (Stolt et al., 2012) a method using no sensors or knowledge of the dynamics is presented. It uses only the error between the desired and actual joint angle. An assumption is made that this error is proportional to the joint torques. We will show that this estimate can be improved if an inverse dynamics controller, such as the one presented in chapter 4, is used. The force estimate is based on the closed-loop dynamics of the system and controller. Our method will be implemented in simulations, and the efficiency of the force estimate will be discussed from the results in chapter 7.

5.1. Dynamics and Collision Forces

When an external force is applied on the robot manipulator, the robot dynamics (3.1) becomes augmented with an additional term,

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = \tau + \tau_K \tag{5.1}$$

where τ_K is the induced torque in the manipulator's joints due to external force. The relationship between torque and end-effector force is given by (Spong et al., 2006, sec. 4.10)

$$\tau_K = J^T F$$

where $J = J(q)$ is the Jacobian as defined in appendix A.1, and $F = [F_x,\, F_y,\, F_z,\, n_x,\, n_y,\, n_z]^T$ represents the vector of external forces and moments acting at the end-effector. For a six-dof manipulator, such as the UR5, the force can be found from

$$F = \left(J^T\right)^{-1} \tau_K \tag{5.2}$$

as long as J is not singular.

5.2. Force Estimate from Joint Angle Error

In (Stolt et al., 2012) the error between the measured joint position q and desired joint position q_d is assumed to be proportional to the joint torque. Little mathematical justification is made for this claim, but an experiment was able to provide a decent estimate of the force. Our investigation will show that this method can be improved if an inverse dynamics controller such as presented in chapter 4 is used by the manipulator and the inertia matrix $M(q)$ is known.

Consider the inverse dynamics controller (4.1) applied to the dynamics (5.1)

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = M(q)a_q + C(q,\dot{q})\dot{q} + g(q) + \tau_K \tag{5.3}$$

where $a_q = \ddot{q}_d - K_0\tilde{q} - K_1\dot{\tilde{q}}$ as in (4.4). Left-multiplying by the inverse of $M(q)$ and removing equal terms leads to the dynamics

$$\ddot{q} = a_q + M(q)^{-1}\tau_K \tag{5.4}$$

Since $M(q)$ is positive definite, it is always invertible. Inserting for a_q results in the closed-loop dynamics

$$\begin{aligned} \ddot{\tilde{q}} + K_1\dot{\tilde{q}} + K_0\tilde{q} &= M(q)^{-1}\tau_K \\ &= r \end{aligned} \tag{5.5}$$

where r is defined such that

$$r \triangleq M(q)^{-1}\tau_K \tag{5.6}$$

Note that r acts like a force on the mass-spring-damper system $\tilde{q}$. In (Hacksel and Salcudean, 1994) a velocity observer leads to similar dynamics as in (5.5), and they propose to estimate the right-hand side such that

$$\hat{r} = K_0\tilde{q} \tag{5.7}$$

where $\hat{r}$ is the estimate of r. If r is slowly varying compared to the dynamics of $\tilde{q}$, this should give a good estimate. The dynamics of $\tilde{q}$ is dependent on the chosen K_0 and K_1 matrices, and in theory they could be chosen such that the system is very quick (stiff). But in practice, the system will be limited by motor torque saturation, reaction time, and possibly other disturbances.

By using (5.2), (5.7), and the torque estimate $\hat{\tau}_K = M(q)\hat{r}$, the estimated force $\hat{F}$ can now be found

$$\begin{aligned} \hat{F} &= \left(J^T\right)^{-1}\hat{\tau}_K \\ &= \left(J^T\right)^{-1} M(q)K_0\tilde{q} \end{aligned} \tag{5.8}$$

This force estimate is implemented in our simulations. Whether the assumption that r is slowly varying gives a good estimate will be discussed after presenting the simulation results in chapter 7.

Similarities exist between this force estimate and the method used in (Stolt et al., 2012). If we dropped $M(q)$ from (5.8), and exchanged K_0 for a new experimentally determined constant, we would have the same estimator as in (Stolt et al., 2012). By taking advantage of the closed-loop dynamics of the inverse dynamics controller, we have shown that a better estimate can be achieved by taking into account the configuration-dependent inertia matrix.

5.3. Collision Detection from Force Estimate

An estimate of the external contact force is given by (5.8). A collision is registered when the magnitude of the force is above some limit, i.e.

$$\left\| \hat{F} \right\| > f_{min} \tag{5.9}$$

where $\|\cdot\|$ is the Euclidean norm, and f_{min} is a positive constant. As we only make use of the magnitude of the force, the force estimate does not need to be accurate in all directions, as long as the magnitude is in the correct range.

In our simulation experiment, the detected collision will be signaled to the collision avoidance system presented in the next chapter.

Chapter 6

Collision Avoidance and Trajectory Generation

The goal of this chapter is to develop a method to dynamically generate a collision-free trajectory to the manipulator's destination. This trajectory will be updated whenever a collision with an object is detected using the method from the previous chapter. The overall strategy is to

1. Start moving from initial position P_0 to destination P_d
2. Whenever a collision occurs, update collision map and create a new trajectory around all detected objects

Several methods exist for generating collision-free trajectories. The problem can be divided into two parts. First, waypoints are constructed in workspace around any detected objects which takes the manipulator from the current position P to P_d. Then the waypoints are mapped to the configuration space, and a trajectory between each waypoint is generated. The trajectory in configuration space is interpolated using smooth splines, such that a complete time-set of q_d, $\dot{q}_d$, and $\ddot{q}_d$ is generated all the way to the destination. This desired trajectory will be fed to the inverse dynamics controller, which will move the manipulator toward the target position.

In (Spong et al., 2006), two approaches are presented for generating waypoints. The first method constructs artificial potential fields and uses a gradient descent algorithm to find a path. The second is called the probabilistic roadmap method and uses random samples to find collision-free points, which are then connected to their neighbors. Then a path is found using the shortest path through all the points.

We will take a different approach to generating the waypoints by using a Hopfield-type neural network as presented by (Glasius et al., 1995). They propose a method by placing neurons on an n-dimensional grid, each neuron representing a point in the n-dimensional space. Each of these is connected to their neighbors, and their state is updated iteratively by using the states of the neighboring neurons. Using Lyapunov theory, convergence is proven for the network, and the generated path is shown to be minimal.

Waypoints could be generated directly in configuration space to avoid collision in all links of the manipulator. But since the configuration space of the UR5 manipulator is 6-dimensional, the neural network quickly becomes very complex. Thus, we will keep it simple by generating waypoints in workspace. A further simplification is made such that the end-effector only moves in a two-dimensional plane. This enables us to use a two-dimensional collision space. It should be noted that these simplifications limits us to only generate collision-free paths at the end-effector, as opposed to every link on the manipulator.

6.1. Hopfield Neural Network Path Planner

In chapter 2 an overview of neural networks was presented. A Hopfield network was briefly discussed, and a type of this network will be presented in full detail here. A proof based on Lyapunov theory will be presented for stability and convergence of the network. It will be shown that this network can be used to provide a solution to the shortest-path problem. The Hopfield model is based on the neural network in (Glasius et al., 1995).

6.1.1. Dynamics

The following proceedings are based on (Glasius et al., 1995) and will present the neural network and show its stability. Let us denote the workspace of the manipulator by $\mathcal{W} \subset \mathbb{R}^3$. Within a two-dimensional plane in the workspace, a set of topologically ordered neurons is evenly distributed. This map will be called the neuronal space $\mathcal{N} \subset \mathbb{R}^2$.

Denote the output of each neuron as σ_i, $i = 1, \ldots, N$, where N is the total number of neurons. The total input to each neuron i is

$$u_i(t) = \sum_{j}^{N} T_{ij}\sigma_j(t) + I_i \tag{6.1}$$

where the synaptic weight between neuron i and neuron j is represented by T_{ij}, and I_i is an external sensory input. The destination neuron is clamped to 1 by setting $I_i \gg 1$. Neurons located on obstacles have an external sensory input $I_i \ll$ -1, such that these neurons will be clamped to 0. The connections of T_{ij} is defined such that

$$T_{ij} = \begin{cases} 1 & \text{if } \rho(i,j) < r \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

where r is a positive constant, and $\rho(i,j)$ is the Euclidean distance between neuron i and j in $\mathcal{N}$. The discrete-time update law for the neurons is given by

$$\sigma_i(t+1) = g\left(u_i(t)\right) \tag{6.3}$$

where $g(\cdot)$ is an activation function such as those presented in chapter 2. Consider the

Lyapunov function candidate

$$L(\sigma) = \frac{1}{2}\sum_{i,j} T_{ij}\sigma_i\sigma_j - \sum_i I_i\sigma_i + \sum_i G(\sigma_I) \tag{6.4}$$

with $G(\sigma_i) = \int_0^{\sigma_i} g^{-1}(x)dx$. The time-step difference of the Lyapunov function candidate is found to be

$$\begin{aligned} \Delta L &= L(\sigma(t+1)) - L(\sigma(t)) \\ \Delta L &= -\frac{1}{2}\sum_{i,j} T_{ij}\left[\Delta\sigma_i\Delta\sigma_j + 2\sigma_j(t)\Delta\sigma_i\right] - \sum_i I_i\Delta\sigma_i \\ &\qquad + \sum_i \left[G\left(\sigma_i(t+1)\right) - G\left(\sigma_i(t)\right)\right] \end{aligned} \tag{6.5}$$

where $\Delta\sigma_i = \sigma_i(t+1) - \sigma_i(t)$. By Taylor's theorem

$$\begin{aligned} G\left(\sigma_i(t+1)\right) - G\left(\sigma_i(t)\right) &= G'\left(\sigma_i(t+1)\right)\Delta\sigma_i - \frac{1}{2}G''(\xi)(\Delta\sigma_i)^2 \\ &\leq G'\left(\sigma_i(t+1)\right)\Delta\sigma_i - \frac{1}{2}\left(\Delta\sigma_i\right)^2 \min_{x\in[0,1]} G''(x) \end{aligned} \tag{6.6}$$

where $\xi \in [\sigma_i(t), \sigma_i(t+1)]$. Inserting (6.6) into (6.5) and using the definition of $G(x)$ we obtain

$$\Delta L \leq -\frac{1}{2}\sum_{i,j}\left[T_{ij} + \frac{\delta_{ij}}{\beta}\right]\Delta\sigma_i\Delta\sigma_j \tag{6.7}$$

where $\frac{1}{\beta} = \min_x G''(x)$ is the minimum curvature of the function $G(x)$ in the closed interval [0, 1]. The Kronecker delta-function δ_{ij} is given by

$$\delta_{ij} = \begin{cases} 0, & \text{if } i \neq j \\ 1, & \text{if } i = j \end{cases}$$

Consider the matrix $T \in \mathbb{R}^{N\times N}$ made up from the elements T_{ij}. The right-hand side of (6.7) will be negative if the matrix $T + \beta^{-1}I$ is positive definite. A sufficient condition for $T + \beta^{-1}I$ to be positive definite is given by

$$\beta < \left|\frac{1}{\lambda_{\text{min}}}\right| \tag{6.8}$$

where λ_{min} is the most negative eigenvalue of T. If this condition is satisfied, L is a Lyapunov function and guarantees that the dynamics converges to a fixed point.

We have shown that the time-discrete dynamics is stable by following the proof of (Glasius et al., 1995). In their paper, they also provide proof for continuous-time version of the dynamics which is omitted here.

6.1.2. Shortest Feasible Path

The stability of the neural network was shown in the previous section. In this section we will first provide conditions for which the equilibrium point is unique. Then it will be argued that the neural network provides the shortest feasible path using the steepest ascent in terms of σ from the initial neuron to the destination neuron.

The final equilibrium states are solutions of the fixed point equations

$$\sigma_i^* = g\left(\sum_j^N T_{ij}\sigma_j^* + I_i\right) \tag{6.9}$$

for $i = 1, \ldots, N$. This final equilibrium point is unique when the Lyapunov function is strictly convex. If the second-order partial derivatives of L exists then L is strictly convex iff the Hessian matrix $L_{ij} \equiv \frac{\delta^2 L}{\delta\sigma_i \delta\sigma_j}$ is positive definite. The Hessian is found to be

$$\begin{aligned} L_{ij} &= -T_{ij} + \delta_{ij}\frac{1}{g'\left(g^{-1}(\sigma_i)\right)} \\ &< -T_{ij} + \frac{\delta_{ij}}{\beta} \end{aligned} \tag{6.10}$$

If all eigenvalues of T are negative, then L is strictly convex. If there are some positive eigenvalues, a sufficient condition is given by

$$\beta < \frac{1}{\lambda_{\max}} \tag{6.11}$$

To satisfy (6.8) and (6.11) at the same time, we can choose

$$\beta < \frac{1}{\lambda}, \quad \lambda = \max\left\{|\lambda_{\min}|, |\lambda_{\max}|\right\} \tag{6.12}$$

This choice will then guarantee strict convexity of L, which means that a local minimum of L is a global one and is unique. Thus, (6.9) will only have one solution. This equilibrium state depends only on the position of the obstacles and on the position of the target.

In (Glasius et al., 1995) it is argued that the step giving the shortest path toward the destination from any given neuron, is given by the largest value σ_i of any neighboring neuron i. This means that to construct an optimal path, each step from the initial point must follow the steepest ascent. The destination is reached when no neighboring neurons are larger than the current.

6.1.3. Comparison to other Neural Networks

The neural network presented here is unique from many other neural networks in several regards. First of all, no information is stored in the synaptic weights. They are constant and either one or zero. The only information comes from the external input signals in terms of destination and obstructions. The neural network here is in fact only used to calculate the given problem, and not to learn, store information, or retrieve information as is common in other types of neural networks.

6.2. Path Planner Examples

A few simple examples are simulated to show how well the path planner performs.

6.2.1. Set-up

The Hopfield neural network path planner is implemented in Matlab, and run on several different collision maps. All collision maps are 2-dimensional, with a grid size of 20×20, which results in $N = 400$ neurons. The radius r in (6.2) is set such that only the four nearest neighbors of the current neuron is connected. This means that for each step, the trajectory can only move up/down, or left/right. By solving the eigenvalues of T, we have $\lambda \approx 3.0$, and using (6.12), we have the condition $\beta < \frac{1}{3.0}$. We chose $\beta = 0.1$. The Matlab code of our implementation is seen in appendix C.2.3.

6.2.2. Results and Observations

The generated paths are shown in Figure 6.1. The paths generated are indeed optimal, although only in the sense of the 1-norm between the neurons. This leads to some path segments with an "L"-shape, and others with a "staircase"-shape. These unwanted shapes could potentially be smoothed in a post-processing step.

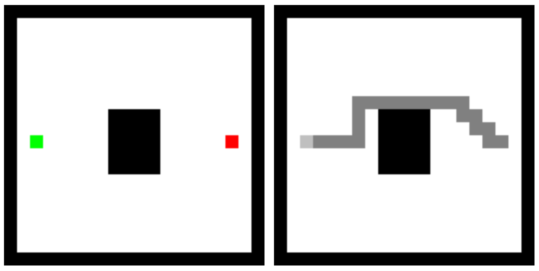
This set-up, with the same number of neurons, will be used for the final simulation in chapter 7. The path planner will then run each time a collision is detected, and a new trajectory will be generated.

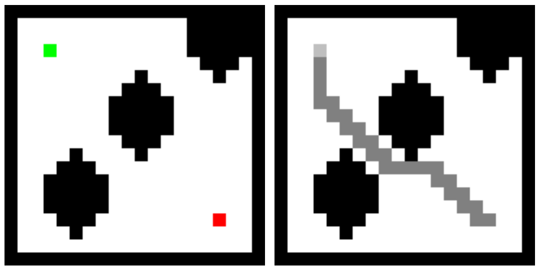
6.3. Trajectory between Waypoints

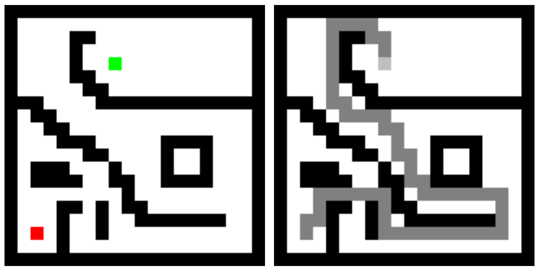
The neural network path planner presented in the previous sections provides waypoints in the neuronal space, but the inverse dynamics controller will need a smooth trajectory in configuration space. To accomplish this, first, the waypoints need to be mapped to the workspace of the manipulator. This mapping will depend on the location of the neuronal space in the workspace. The workspace waypoints are then mapped to configuration space using *inverse kinematics*. Lastly, the configuration space waypoints are interpolated using smooth splines.

6.3.1. Waypoints to Workspace

The neuronal space can be placed at any desired location in the workspace. The mapping from neuronal space to workspace is thus dependent on the chosen placement. In our


(a) Collision map 1


(b) Collision map 2


(c) Collision map 3

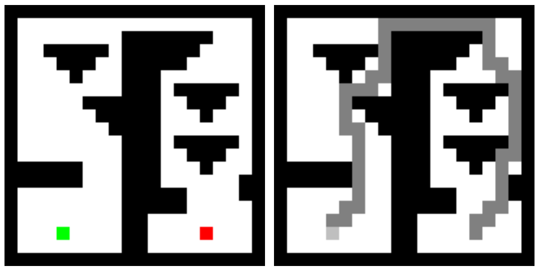

(d) Collision map 4

Figure 6.1: Path planner run on four different collision maps. For each pair, the left image shows the initial set-up, and the right image shows the path generated from the path planner. *White:* Free space, *Black:* Obstruction, *Grey:* Generated path, *Green:* Initial position, *Red:* Destination.

case, the 2-dimensional neuronal space is placed along the xy-axes of the workspace with a mapping such that

$$\begin{aligned} x_w &= 0.0263u + 0.1238 \\ y_w &= 0.0263v - 0.2762 \\ z_w &= 0 \end{aligned} \tag{6.13}$$

where x_w, y_w, and z_w are the three Cartesian coordinates of the workspace, and u, v are the two integer coordinates of the neuronal space. The first neuron is defined to be located at $(u, v) = (1, 1)$ and the last one is located at $(u, v) = (m, n)$, where m and n are the number of neurons along the u-axis and v-axis respectively. Since the neuronal space is evenly distributed in the collisions space, the same mapping as in (6.13) can be used for collision space. Moreover, the collision space coordinates can use any real value, not just integers.

6.3.2. Waypoints to Configuration Space

The workspace waypoints can be mapped to the configuration space using inverse kinematics. The topic of inverse kinematics is a very large topic in itself, and will not be presented in any detail here. In our simulations, the inverse kinematics solution from the robotics toolbox (Corke, 2011) has been used. The toolbox provides a function which maps the workspace waypoints to the configuration space.

6.3.3. Connecting Waypoints using Smooth Splines

The inverse dynamics controller needs a smooth trajectory in the configuration space, q_d, $\dot{q}_d$, and $\ddot{q}_d$. To connect the configuration space waypoints, we will use smooth splines. A smooth spline is simply a polynomial, and we will use a polynomial of the form

$$q_d(t) = At^5 + Bt^4 + Ct^3 + Dt^2 + Et + F, \quad t \in [t_0, t_1) \tag{6.14}$$

with the coefficients found such that $q_d(t_0) = q_0$, $q_d(t_1) = q_1$, $\dot{q}_d(t_0) = \dot{q}_0$, $\dot{q}_d(t_1) = \dot{q}_1$, and possibly $\ddot{q}_d(t_0) = \ddot{q}_d(t_1) = 0$. The first and second time-derivative of (6.14) can be calculated in order to find $\dot{q}_d(t)$ and $\ddot{q}_d(t)$.

By providing a list of $(t_0, t_1, q_0, q_1, \dot{q}_0, \dot{q}_1)$ between each waypoint, and using (6.14), all the splines can be connected to form the complete time-set of q_d, $\dot{q}_d$, and $\ddot{q}_d$ until the destination is reached.

In our simulations, the polynomial in (6.14) is calculated using the robotics toolbox (Corke, 2011). The splines are connected and the resulting q_d, $\dot{q}_d$, and $\ddot{q}_d$ are output to the inverse dynamics controller.

Chapter 7

Simulation Experiment

In the preceding chapters, we have developed all the necessary components needed to simulate our proposed solution. We have developed the dynamics of the UR5 manipulator, an inverse dynamics controller, a method for collision detection, and finally a method for collision avoidance of detected objects. All of these components will be combined and used in this simulation experiment.

The goal of the simulation experiment is for the manipulator to move from the initial position P_0 to the destination P_d without any knowledge of the environment. When the manipulator collides into an object, the manipulator will be reversed, and a new trajectory generated around the detected collision point. This process is repeated until the destination P_d is reached.

First we will show some of the implementation details and set-up of the simulation, then results will be presented, and finally the efficiency of the proposed solution will be discussed.

7.1. Set-up

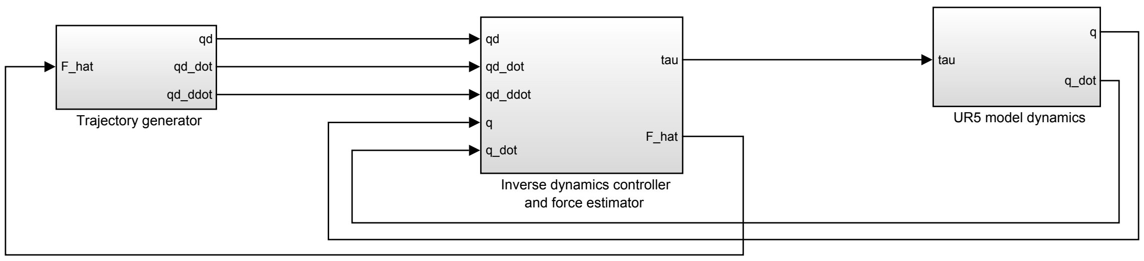


Figure 7.1: Top level signal flow of the implemented Simulink model.

The UR5 manipulator dynamics, collision detection, and trajectory generator are all implemented in Simulink. The top level signal flow is shown in Figure 7.1. Each of the subsystems in this diagram can be seen in Appendix C, along with some of Matlab functions used in the simulation.

The 2-dimensional collision space has been placed in the xy-plane of the workspace as defined by the mapping (6.13). The start and destination positions, P_0 and P_d, are both located in the collision space, and the trajectory generator will restrict all desired trajectories to this plane.

The natural frequencies of the inverse dynamics controller used in (4.6) has been set to

$$\omega = \begin{bmatrix} 22.5 & 22.5 & 37.5 & 60.0 & 67.5 & 75.0 \end{bmatrix}^T$$

The threshold of the collision detection in (5.9) has been set to

$$f_{min} = 12$$

The initial position of the end-effector and destination point in the collision space can be seen in Figure 7.2 together with the objects the manipulator will collide with.

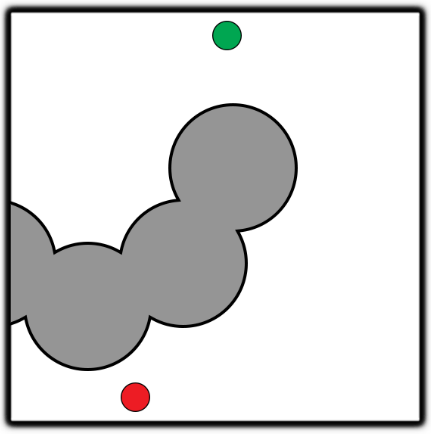


Figure 7.2: Collision space set-up. *Grey* area is collidable objects made up of overlapping cylinders, *green* circle is initial position, and *red* circle is destination.

Objects and Collision Force

Several cylinders have been placed in the collision space, which generates contact forces when the manipulator collides into them. As mentioned in chapter 6, only the end-effector of the manipulator is considered in the collision model. Let us define a vector v relating the position of the cylinder to the end-effector in the xy-plane as

$$v \triangleq \begin{bmatrix} P_{cyl}(1) - P(1) \\ P_{cyl}(2) - P(2) \\ 0 \end{bmatrix}$$

where P_{cyl} is the position of the cylinder's center, and P is the end-effector position, both located in workspace. The parenthesis denotes the element of the vector. Let us denote the distance between the end-effector and cylinder in the xy-plane as D, given by $D = ||v||$. The contact force is then modeled as

$$F_{contact} = \begin{cases} -k\,(R-D)\,\frac{v}{D} & \text{if } D < R \\ 0 & \text{else} \end{cases}$$

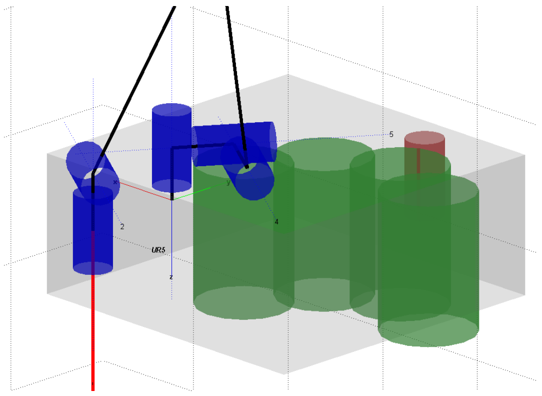
where R is the cylinder radius and k is a constant set to 10^5 in our simulation. Essentially, the contact force increases linearly as the end-effector moves toward the center of the cylinder, starting with zero at the edge. The cylinder thus has a certain softness to it controlled by the constant k, the higher the value, the stiffer the material.

The contact force of all the cylinders are summed up and then applied to the end-effector. We have not implemented any friction force in this simulation.

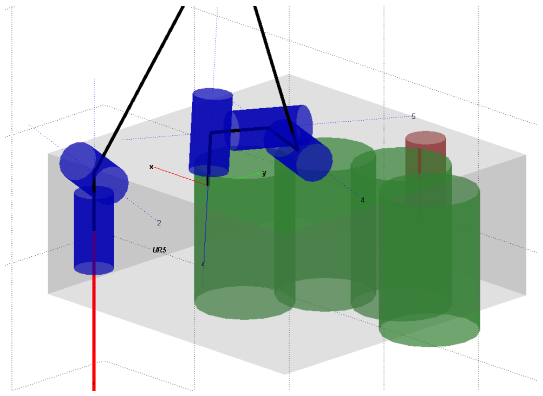
7.2. Results

The simulation was run in Matlab/Simulink, and several plots and figures are presented to provide results for each of the components of the proposed solution. The simulation was run for 50 seconds until the destination point was reached. The movements of the manipulator is plotted in a 3d-animation using the robotics toolbox (Corke, 2011), and snapshots of this animation can be seen in Figure 7.3. Each time a collision was detected a new collision map was generated, and the evolution of this map can be seen in Figure 7.4. In Figure 7.5 we can see the actual trajectory of the end-effector of the manipulator in collision space, plotted with the final collision map.

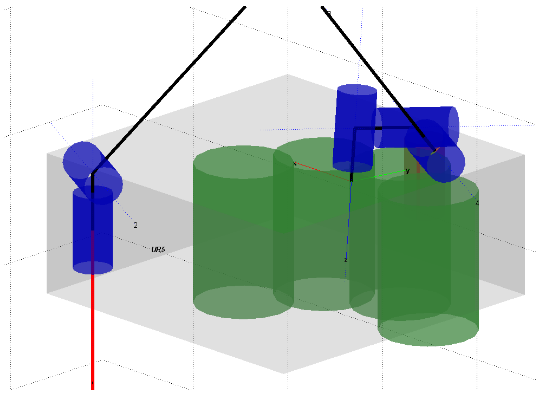
The force estimate for the collision detection scheme is plotted in Figure 7.6 together with the actual collision force. The magnitude of the estimated and actual force is plotted in Figure 7.7 with the number of collisions detected. A zoomed-in extract of the force magnitudes during a collision is shown in Figure 7.8. The joint angle error $\tilde{q}$ is plotted in Figure 7.9.



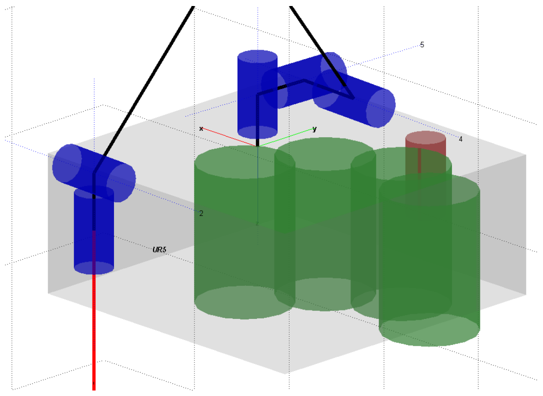
(a) $t = 0\text{s}$



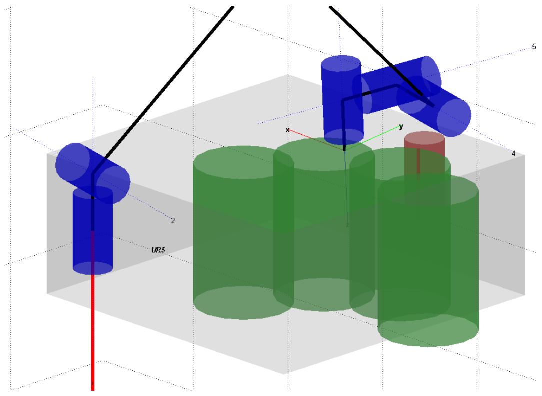
(b) $t = 3\text{s}$



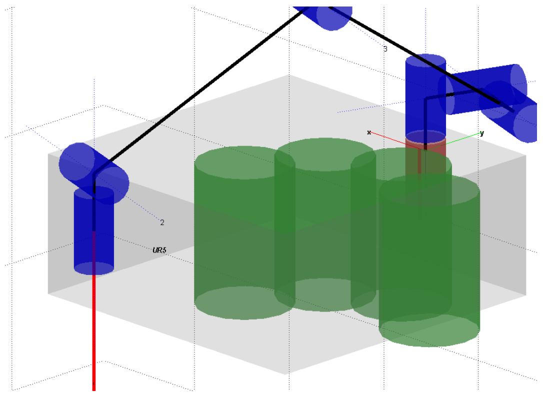
(c) $t = 14\text{s}$



(d) $t = 37\text{s}$



(e) $t = 41.5\text{s}$



(f) $t = 46\text{s}$

Figure 7.3: Snapshots from animation as the manipulator moves toward the destination while exploring the environment. Blue cones represent the manipulator joints, green cones are collidable objects, and the red cone is the destination. The black grid is surrounding the collision (neuronal) space. **(a)** Initial position, **(b)** first collision, **(c)** another collision, **(d)** moving freely, **(e)** collision not registered, **(f)** destination reached.

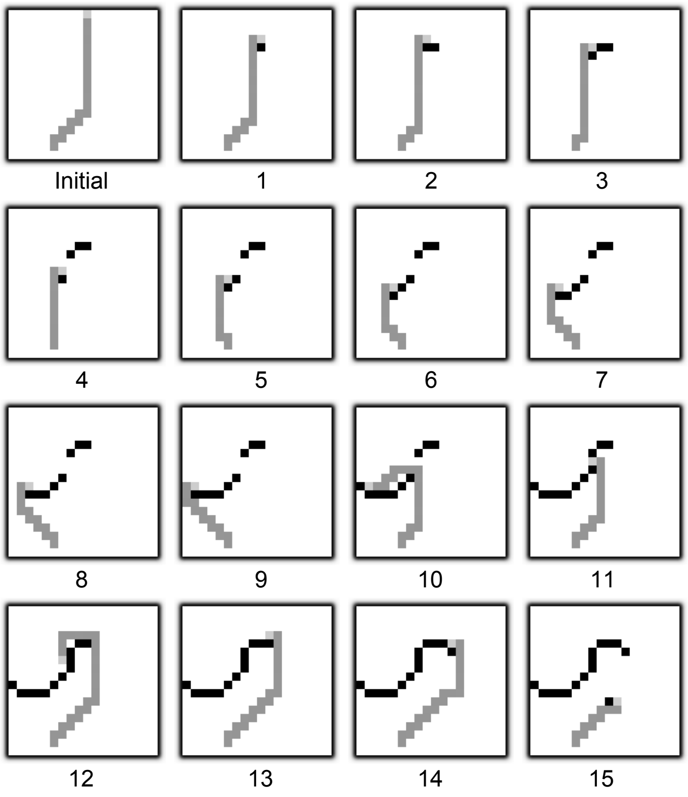


Figure 7.4: Generated collision map and new trajectory after every collision. *White:* Free space, *Black:* Detected obstruction, *Light grey:* Current position, *Dark grey:* Generated path

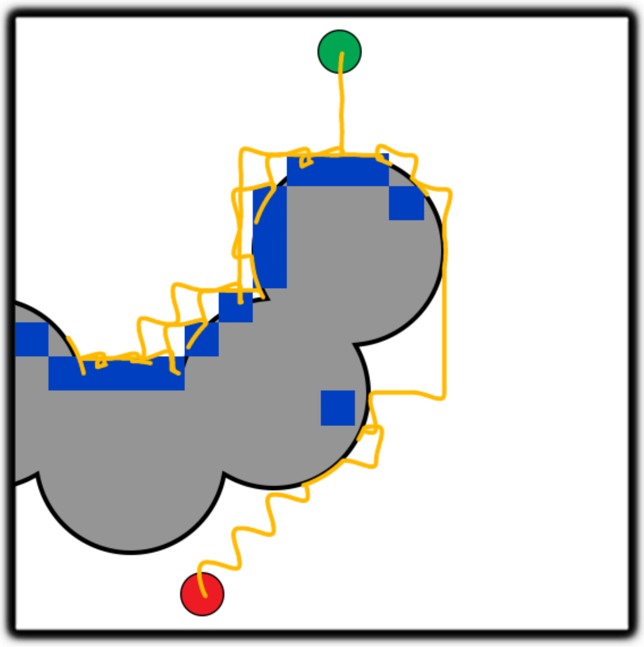
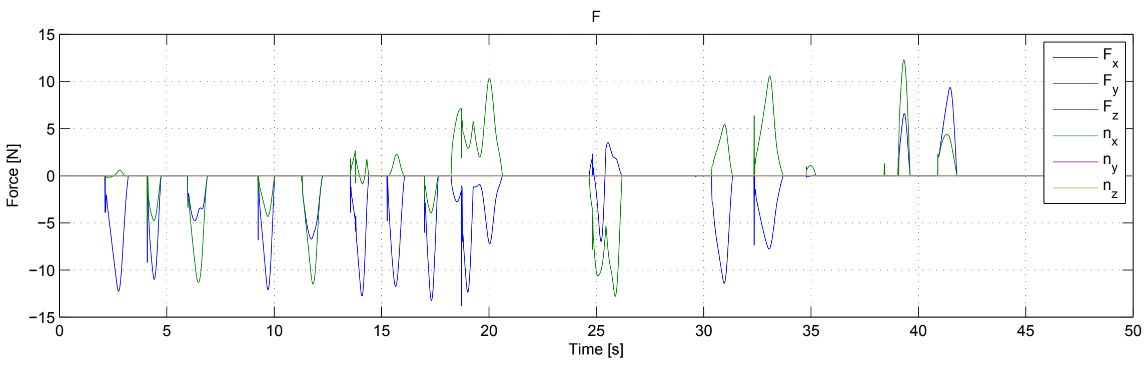


Figure 7.5: Actual trajectory of manipulator end-effector (*orange*) in collision space, superimposed with final generated collision map (*blue*).



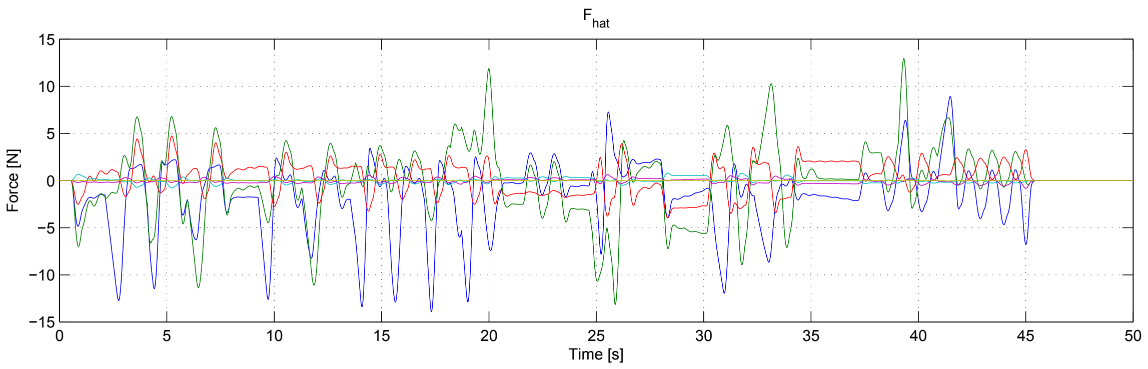


Figure 7.6: Actual external force F and estimated external force $\hat{F}$.

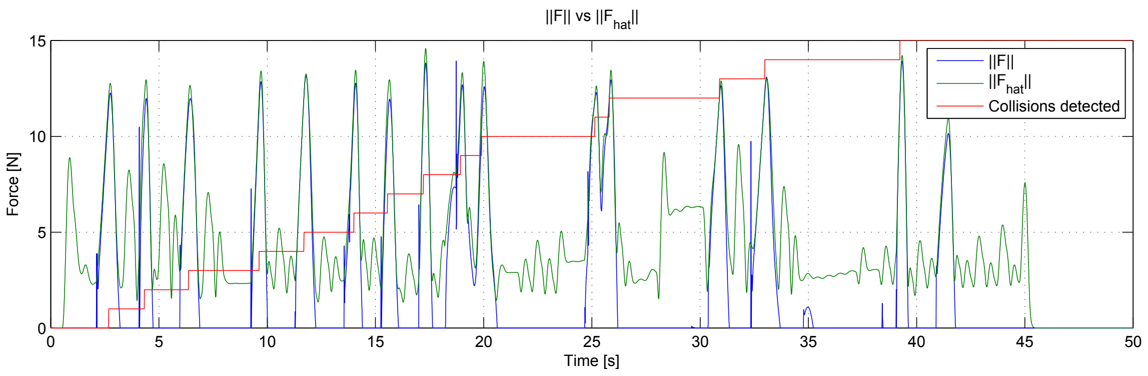


Figure 7.7: Magnitude of the actual and estimated external force, $||F||$ and $||\hat{F}||$, plotted with the total number of detected collisions.

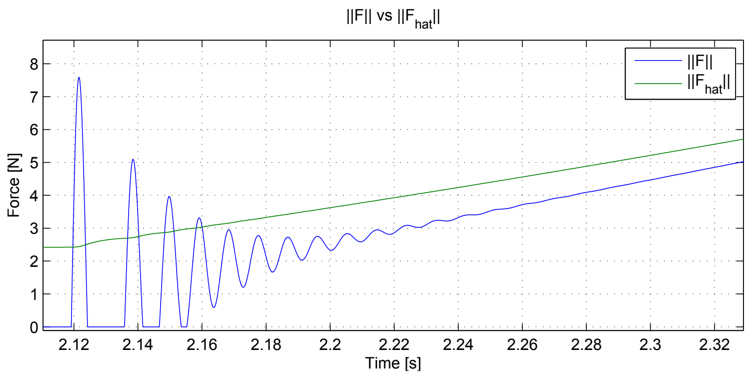


Figure 7.8: Zoomed-in section of the actual and estimated force magnitudes, $||F||$ and $||\hat{F}||$, showing high-frequency forces during a collision.

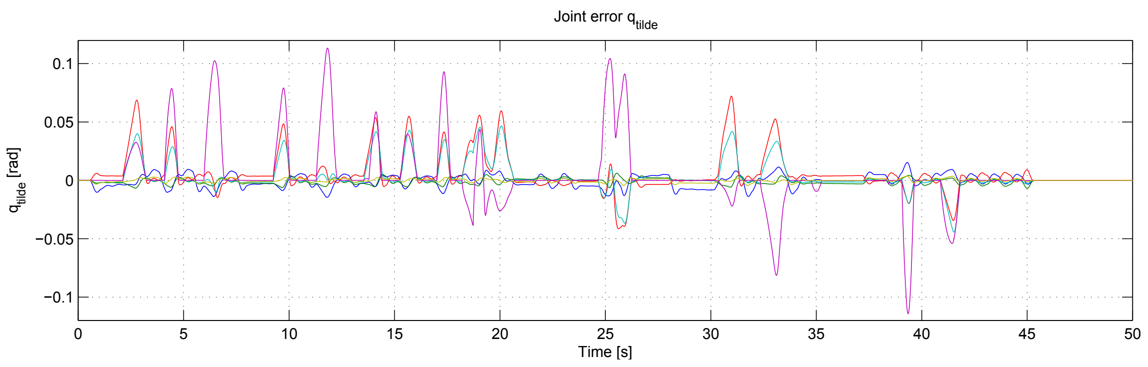


Figure 7.9: Joint angle error $\tilde{q}$.

7.3. Discussion

The efficiency of the collision avoidance and trajectory generator will be discussed first, before we continue with the collision detection scheme. Finally, we will reflect on the effectiveness of the overall system.

Collision Avoidance and Trajectory Generator

The collision map starts out empty so a direct trajectory is generated from P_0 to P_d as seen in Figure 7.4. As the manipulator moves and new collisions are registered, the collision map fills up as expected until the entire left half is blocked. Finally the manipulator ends up at the desired position.

As seen in Figure 7.5, the collisions are sometimes registered inside the objects instead of at the edge. This is because the collision is registered at the point the end-effector is desired to be, rather than it's actual position. This, combined with some delay from the collision detection scheme makes the collision become registered inside the object. Using the actual position of the end-effector proved challenging, primarily for two reasons:

1. The force estimate will fluctuate after a collision has been detected, which may result in repeatedly detected collisions at the same position. This happens because the new trajectory will bring the error $\tilde{q}$ to zero at first, but the manipulator may still be affected by collision forces, so when $\tilde{q}$ increases again a new collision will be registered.

2. The manipulator may get stuck in a loop. Consider the case when a collision is first detected, and a new trajectory is generated around it. If the new trajectory takes the manipulator to a collidable object close by, the end-effector might end up sliding along the object all the way back to where it first collided because of the delay in the collision detection. Then the collision will be detected at the same point as the first time, and the same trajectory is generated. Then it starts over again getting stuck in an infinite loop.

Both of these problems are eliminated by using the desired joint position q_d as a starting point for the collision detection and trajectory generator, at the expense of a less accurate collision map as has been done in our simulations. These problems can possibly be dealt with in other ways, and may be a subject of further study.

Consider again Figure 7.5 and notice the zigzag motion of the trajectory especially noticeable near the destination point. This happens because waypoints are placed by following the zigzag trajectories seen in Figure 7.4. As discussed in chapter 6, the waypoints could be placed in such a way that the trajectory becomes smoother. Exactly how this would be done is another subject of further study.

Collision Detection

The accuracy of the collision detection scheme is dependent on the external force estimate. As can be seen from Figure 7.6 and Figure 7.7, the estimated collision force is fairly accurate most of the time. But the estimated force has a tendency to register forces when they are not occurring, and this seems to be extra dominant whenever the manipulator moves violently, such as in the zigzag-motion at the end, and during the acceleration at the very beginning. Such rapid movement produces an error in terms of $\tilde{q}$ in the order of 10^{-3} [rad] as seen in Figure 7.9, which leads to an error in the force estimate. With perfectly known dynamics, and no collision forces, the error should have been zero. But the non-zero error is thought to be caused by numerical limitations in the simulation and limited resolution on the desired trajectory q_d and its derivatives.

Because of the assumption of a slowly varying r in (5.5), the high-frequency components of the collision-induced torque is removed from the estimate, and thus also from the force estimate. This is evident in Figure 7.8. As the manipulator collides with an object, the energy of the manipulator is usually dissipated in a very short time-span. Thus, the initial collision is not registered by this collision detection scheme, and it's only when the inverse dynamics controller starts to slowly increase the applied force due to the increasing $\tilde{q}$ that the system will detect collision. This delay is in the order of 0.2-0.5 seconds, although these values are expected to vary with the chosen ω of the controller.

In fact, the force estimation method seems to perform very well as long as the manipulator is not making violent movement, and as long as high-frequency components are not needed. On the other hand, for collision detection, this method does not prove adequate. For future studies, it is recommended to look into other methods to detect collisions and see if these can provide quicker response time. Suggestions for such methods could be the energy-based method of (De Luca et al., 2006), although this requires a different formulation of the manipulator dynamics. One could also consider making a disturbance observer as in (Takakura et al., 1989), or try to use $\tilde{q}$ and $\dot{\tilde{q}}$ directly for collision detection.

The force estimation method should be tested on a real manipulator to investigate its effectiveness. In this simulation, it is assumed that dynamics are perfectly known and no joint friction. With such effects included, the estimation is assumed to become less accurate.

Overall Scheme

The manipulator can be seen to reach its destination without any prior knowledge on the environment, and as such the experiment is considered a success. However, the simulation did show that improvements can be made both to the collision avoidance and collision detection scheme.

This simulation limited the collision space to a 2-dimensional plane of the workspace. The methods presented here should be easily generalized to a 3-dimensional collision space. An even more generalized system should be able to handle collisions along all links in the manipulator. For such a system, the collision space would need to be placed in the 6-dimensional configuration space of the manipulator. Since the number of neurons needed in the collision avoidance scheme increases exponentially with number of dimensions in the neuronal space, the system will quickly become complex. Using 20 neurons along each dimension as we used in our 2-d scheme, a 6-d collision space will require $20^6 = 6.4 \cdot 10^7$ neurons. Increasing the grid resolution in 6-d will quickly demand impossible amount of computing power. Another important problem that would need to be solved for a 6-d collision space is the mapping of objects detected in workspace to the 6-d collision space.

For implementation in a real manipulator, the scheme must be able to run in real-time. The 50 sec of time simulated, used around 20 min of computational time, which is obviously too long for real-time. But the dynamics of the manipulator would not need to be calculated in a real-time system, which would reduce the computation time. The inverse dynamics controller will still need to be calculated, but is less demanding because it doesn't need to calculate the inverse of the inertia matrix. It was also observed that several seconds elapsed every time a collision was detected, most of the time spent calculating inverse kinematics. Several manipulators, including the UR5 manipulator, use a hardware implementation of the controller and inverse kinematics. The only demanding thing to calculate would then be the neural network path planner, whose complexity is already discussed. Thus, the real-time limitation on this scheme will mostly depend on the manipulator and the complexity of the path planner.

Variations on the scheme could be imagined. Instead of retracting the end-effector and going around the detected object, one could instead go into a force control mode and explore the object by sliding along it. This may lead to a faster and more accurate exploration.

Chapter 8

Conclusion

We have presented methods for detecting collisions in a robot manipulator, and generating new trajectories around all objects we collide into. These methods are combined to take the end-effector of the manipulator from a start point to a destination point in an unknown environment. The methods does not require any external sensors, but can easily be combined with such for better performance.

The proposed solution was run in a simulation using Matlab/Simulink. The simulation experiment was set-up such that the manipulator had to move to a destination point, with collidable objects modeled as cylinders blocking the path. These objects were initially unknown to the manipulator, and the manipulator was shown to successfully navigate the environment and reach its destination.

The simulation and controller required known dynamics of a manipulator, and we developed the dynamics based on the UR5 manipulator. The dynamics were shown to give sensible results when run in a simple simulation.

For the collision detection scheme an original method for force estimation without force sensor was developed by taking advantage of the closed-loop dynamics of the controller. A collision was registered when the norm of the estimated force was over a certain threshold. The method presented was shown to give a pretty accurate result for the force estimation, although only for low-frequency components. High-frequency components of forces were not detectable, such as at initial collision with an object. The collision was only detected after the controller had slowly increased the force applied on the object. Hence, the scheme used an unnecessarily long time before a collision was registered. The force estimation scheme should be investigated further, but is not ideally suited for collision detection.

The collision avoidance scheme used a path planner based on a Hopfield neural network. Such a network was chosen primarily to gain experience with neural networks, an important topic within the field of robot learning. For future studies other implementations should also be considered. The path planner was shown to provide an optimal path and worked fine in the simulation experiment. The path planner generated waypoints in collision space which were mapped to configuration space, and then connected using smooth splines to form a complete trajectory. The trajectory could then be used by the inverse dynamics controller.

8.1. Recommendations for Future Work

In our work, the proposed solution was only implemented in a simulation. A natural extension to this work would be to implement and test it on a real manipulator to see how the system copes with more uncertainties. The performance of the force estimate is especially hard to measure in a simulated environment.

The collision avoidance scheme currently works in a 2-dimensional plane. A further improvement would be to generalize the system to the 3-dimensional workspace, or even better to the 6-dimensional configuration space. This will probably require a more efficient path planner than the method presented here. Additionally, this will require a method of mapping objects detected in workspace to the configuration space. The result will be a solution which will enable the manipulator to explore any environment.

We make an assumption that any collision is occurring at the end-effector. For future studies, it would be interesting to estimate at which link of the manipulator the collision occurred, as well as the exact point of impact along this link.

The simulation showed that the use of the force estimate in the collision detection scheme was not optimal for this task. A more efficient collision detection method should be found. Some suggestions were given in the discussion of section 7.3. The force estimate, however, did provide interesting results for low-frequency forces. This may make it very well suited for some applications, such as assembly tasks. Further studies of the force estimate for such applications would be interesting.

Currently, when a collision is detected, the manipulator is backed up before it tries to move around the point of collision. A better solution could possibly be to switch to a force control mode as soon as a collision is detected and then explore the object by sliding along it. Such a method should be considered in future work.

Appendix A

Definitions and Properties

A.1. Geometric Jacobian

The geometric Jacobian $J_i = J_i(q)$ relates the linear and angular velocities of frame i to the manipulator's angular joint velocities,

$$\begin{bmatrix} v_i^0 \\ \omega_i^0 \end{bmatrix} = J_i \dot{q} \tag{A.1}$$

where v_i^0 is the linear velocity of frame i, and ω_i^0 is the angular velocity of frame i. Both are given in the inertial frame, denoted by the superscript 0. Furthermore, the Jacobian can be divided into its linear and angular part such that

$$J_i = \begin{bmatrix} J_{v_i} \\ J_{\omega_i} \end{bmatrix} \tag{A.2}$$

where the relationship between the frame and joint velocities now becomes

$$v_i^0 = J_{v_i} \dot{q} \tag{A.3}$$

$$\omega_i^0 = J_{\omega_i} \dot{q} \tag{A.4}$$

If no i is given, the Jacobian is assumed to be for the end-effector frame, i.e.

$$J = J_n \tag{A.5}$$

where n is the number of degrees-of-freedom. Details of finding the Jacobian for a given manipulator is given in (Spong et al., 2006, sec. 4.6). The end-effector Jacobian of the UR5 manipulator is calculated using the *Robotics Toolbox* from (Corke, 2011).

A.2. Transformation and Rotation Matrices

A set of *basic homogeneous transformations* is given by (Spong et al., 2006)

$$\text{Trans}_{x,a} = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{Rot}_{x,\alpha} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{Trans}_{y,b} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{Rot}_{y,\beta} = \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{Trans}_{z,c} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{Rot}_{z,\gamma} = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The rotation matrices $R_{\cdot,\cdot}$ are the upper left 3×3 elements of $\text{Rot}_{\cdot,\cdot}$, i.e.

$$\begin{aligned} R_{x,\alpha} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \\ R_{y,\beta} &= \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \\ R_{z,\gamma} &= \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

A.3. Skew Symmetry

An $n \times n$ matrix S is said to be skew symmetric if and only if it has the property such that

$$S + S^T = 0 \tag{A.6}$$

The implications of skew symmetry are that the diagonal elements are all zero, and it only has three independent entries. Every skew symmetric 3×3-matrix can be written in the form (Spong et al., 2006)

$$S = \begin{bmatrix} 0 & -s_3 & s_2 \\ s_3 & 0 & -s_1 \\ -s_2 & s_1 & 0 \end{bmatrix} \tag{A.7}$$

Appendix B

UR5 Manipulator Dynamics

B.1. Framework (Maple file)

The framework is used to generate the dynamics symbolically, and is based on the work of (Høifødt, 2011). The Maple file follows on the next page.

Dynamic Model of the UR 5

The outputs will be written to "Ur5Dynamics.m". Note: The output must be modified slightly to produce valid Matlab syntax.

```
> restart :
> with(LinearAlgebra) :
>
> #Defining joint variables
> q := Vector([[q1(t)], [q2(t)], [q3(t)], [q4(t)], [q5(t)], [q6(t)]]) :
> Dq := map(diff, q, t) :
> DDq := map(diff, Dq, t) :
>
> #Defining link length vectors
> r1c1 := Vector([[0, -0.01541, 0.00151]]) :
> r2c2 := Vector([[0.2125, 0, 0.1134]]) :
> r3c3 := Vector([[0.150, -0.00015, 0.0154]]) :
> r4c4 := Vector([[0, 0.0015, 0.0154]]) :
> r5c5 := Vector([[0, 0, -0.0128]]) :
> r6c6 := Vector([[0, 0, 0]]) :
> r01 := Vector([[0], [0.0892], [0]]) :
> r12 := Vector([[-0.425], [0], [0]]) :
> r23 := Vector([[-0.39243], [0], [0]]) :
> r34 := Vector([[0], [0.109], [0]]) :
> r45 := Vector([[0], [-0.093], [0]]) :
> r56 := Vector([[0], [0], [0.082]]) :
> r0c1 := r1c1 + r01 :
> r1c2 := r2c2 + r12 :
> r2c3 := r3c3 + r23 :
> r3c4 := r4c4 + r34 :
> r4c5 := r5c5 + r45 :
> r5c6 := r6c6 + r56 :
>
> #Gravity vector in inertial frame
> g0 := Vector([[0], [0], [-g]]) :
>
> #Rotation matrices
> α_1 := π/2 :
> α_2 := 0 :
> α_3 := 0 :
> α_4 := π/2 :
> α_5 := -π/2 :
> α_6 := 0 :
>
> Rz := (θ) → Matrix([[cos(θ), -sin(θ), 0], [sin(θ), cos(θ), 0], [0, 0, 1]]) :
> Rx := (α) → Matrix([[1, 0, 0], [0, cos(α), -sin(α)], [0, sin(α), cos(α)]]) :
>
```

```
> R01 := MatrixMatrixMultiply(Rz(q[1]), Rx(α1)) :
> R12 := MatrixMatrixMultiply(Rz(q[2]), Rx(α2)) :
> R23 := MatrixMatrixMultiply(Rz(q[3]), Rx(α3)) :
> R34 := MatrixMatrixMultiply(Rz(q[4]), Rx(α4)) :
> R45 := MatrixMatrixMultiply(Rz(q[5]), Rx(α5)) :
> R56 := MatrixMatrixMultiply(Rz(q[6]), Rx(α6)) :
>
> R02 := MatrixMatrixMultiply(R01, R12) :
> R03 := MatrixMatrixMultiply(R02, R23) :
> R04 := MatrixMatrixMultiply(R03, R34) :
> R05 := MatrixMatrixMultiply(R04, R45) :
> R06 := MatrixMatrixMultiply(R05, R56) :
>
> #Axis of rotation of joint i expressed in frame i
> z0 := Vector([[0], [0], [1]]) :
> b1 := MatrixMatrixMultiply(Transpose(R01), z0) :
> b2 := combine(MatrixMatrixMultiply(Transpose(R02), MatrixVectorMultiply(R01, z0)), trig) :
> b3 := combine(MatrixMatrixMultiply(Transpose(R03), MatrixVectorMultiply(R02, z0)), trig) :
> b4 := combine(MatrixMatrixMultiply(Transpose(R04), MatrixVectorMultiply(R03, z0)), trig) :
> b5 := combine(MatrixMatrixMultiply(Transpose(R05), MatrixVectorMultiply(R04, z0)), trig) :
> b6 := combine(MatrixMatrixMultiply(Transpose(R06), MatrixVectorMultiply(R05, z0)), trig) :
>
> #Link Masses
> m1 := 3.8054 :
> m2 := 8.4576 :
> m3 := 2.1456 :
> m4 := 1.1884 :
> m5 := 1.11884 :
> m6 := 0.1632 :
>
> #Sylinder link dimensions
> r1 := 0.045 :
> r2 := 0.045 :
> r3 := 0.045·97/113 :
> r4 := 0.045·97/113 :
> r5 := 0.045·97/113 :
> r6 := 0.045·97/113 :
>
> h1 := 0.13693 :
> h2 := 0.425 + 0.115 :
> h3 := 0.392 + 0.115 :
> h4 := 0.085 :
> h5 := 0.085 :
> h6 := 0.0395 :
```

```
>
> #Inertia matrices
> I_cyl_x := (m, r, h) →Matrix([[1/2·m·r^2, 0, 0], [0, 1/12·m·(3·r^2 + h^2), 0], [0, 0, 1/12·m·(3·r^2 + h^2)]]) :
> I_cyl_y := (m, r, h) →Matrix([[1/12·m·(3·r^2 + h^2), 0, 0], [0, 1/2·m·r^2, 0], [0, 0, 1/12·m·(3·r^2 + h^2)]]) :
> I_cyl_z := (m, r, h) →Matrix([[1/12·m·(3·r^2 + h^2), 0, 0], [0, 1/12·m·(3·r^2 + h^2), 0], [0, 0, 1/2·m·r^2]]) :
>
> I1 := I_cyl_y(m1, r1, h1) :
> I2 := I_cyl_x(m2, r2, h2) :
> I3 := I_cyl_x(m3, r3, h3) :
> I4 := I_cyl_y(m4, r4, h4) :
> I5 := I_cyl_y(m5, r5, h5) :
> I6 := I_cyl_z(m6, r6, h6) :
>
> #Forward recursion: Link 1
> ω1 := b1·Dq[1] :
> α1 := b1·DDq[1] + CrossProduct(ω1, b1·Dq[1]) :
> Dω1 := map(diff, ω1, t) :
> ae1 := CrossProduct(Dω1, r01) + CrossProduct(ω1, CrossProduct(ω1, r01)) :
> ac1 := CrossProduct(Dω1, r0c1) + CrossProduct(ω1, CrossProduct(ω1, r0c1)) :
> g1 := MatrixVectorMultiply(Transpose(R01), g0) :
>
> #Forward recursion: Link 2
> ω2 := combine(MatrixVectorMultiply(Transpose(R12), ω1) + b2·Dq[2], trig) :
> α2 := combine(MatrixVectorMultiply(Transpose(R12), α1) + b2·DDq[2] + CrossProduct(ω2, b2·Dq[2]),
      trig) :
> Dω2 := map(diff, ω2, t) :
> ae2 := combine(MatrixVectorMultiply(Transpose(R12), ae1) + CrossProduct(Dω2, r12) + CrossProduct(ω2,
      CrossProduct(ω2, r12)), trig) :
> ac2 := combine(MatrixVectorMultiply(Transpose(R12), ae1) + CrossProduct(Dω2, r1c2) + CrossProduct(ω2,
      CrossProduct(ω2, r1c2)), trig) :
> g2 := combine(MatrixVectorMultiply(Transpose(R02), g0), trig) :
>
> #Forward recursion: Link 3
> ω3 := combine(MatrixVectorMultiply(Transpose(R23), ω2) + b3·Dq[3], trig) :
> α3 := combine(MatrixVectorMultiply(Transpose(R23), α2) + b3·DDq[3] + CrossProduct(ω3, b3·Dq[3]),
      trig) :
> Dω3 := map(diff, ω3, t) :
> ae3 := combine(MatrixVectorMultiply(Transpose(R23), ae2), trig) :
> ac3 := combine(MatrixVectorMultiply(Transpose(R23), ae2), trig) :
> g3 := combine(MatrixVectorMultiply(Transpose(R03), g0), trig) :
>
> #Forward recursion: Link 4
> ω4 := combine(MatrixVectorMultiply(Transpose(R34), ω3) + b4·Dq[4], trig) :
> α4 := combine(MatrixVectorMultiply(Transpose(R34), α3) + b4·DDq[4] + CrossProduct(ω4, b4·Dq[4]),
      trig) :
```

```
> Dω4 := map(diff, ω4, t) :
> ae4 := combine(MatrixVectorMultiply(Transpose(R34), ae3) + CrossProduct(Dω4, r34) + CrossProduct(ω4,
     CrossProduct(ω4, r34)), trig) :
> ac4 := combine(MatrixVectorMultiply(Transpose(R34), ae3) + CrossProduct(Dω4, r3c4) + CrossProduct(ω4,
     CrossProduct(ω4, r3c4)), trig) :
> g4 := combine(MatrixVectorMultiply(Transpose(R04), g0), trig) :
>
> #Forward recursion: Link 5
> ω5 := combine(MatrixVectorMultiply(Transpose(R45), ω4) + b5·Dq[5], trig) :
> α5 := combine(MatrixVectorMultiply(Transpose(R45), α4) + b5·DDq[5] + CrossProduct(ω5, b5·Dq[5]),
     trig) :
> Dω5 := map(diff, ω5, t) :
> ae5 := combine(MatrixVectorMultiply(Transpose(R45), ae4), trig) :
> ac5 := combine(MatrixVectorMultiply(Transpose(R45), ae4), trig) :
> g5 := combine(MatrixVectorMultiply(Transpose(R05), g0), trig) :
>
> #Forward recursion: Link 6
> ω6 := combine(MatrixVectorMultiply(Transpose(R56), ω5) + b6·Dq[6], trig) :
> α6 := combine(MatrixVectorMultiply(Transpose(R56), α5) + b6·DDq[6] + CrossProduct(ω6, b6·Dq[6]),
     trig) :
> Dω6 := map(diff, ω6, t) :
> ae6 := combine(MatrixVectorMultiply(Transpose(R56), ae5) + CrossProduct(Dω6, r56) + CrossProduct(ω6,
     CrossProduct(ω6, r56)), trig) :
> ac6 := combine(MatrixVectorMultiply(Transpose(R56), ae5) + CrossProduct(Dω6, r5c6) + CrossProduct(ω6,
     CrossProduct(ω6, r5c6)), trig) :
> g6 := combine(MatrixVectorMultiply(Transpose(R06), g0), trig) :
>
> #Backward recursion: Link 6
> f6 := Add(m6·ac6,-m6·g6) :
> τ6 := -CrossProduct(f6, r5c6) + MatrixVectorMultiply(I6, α6) + CrossProduct(ω6, MatrixVectorMultiply(I6,
     ω6)) :
> τ6z := collect(combine(MatrixVectorMultiply(Transpose(b6), τ6), trig), {DDq[1], DDq[2], DDq[3], DDq[4],
     DDq[5], DDq[6], Dq[1], Dq[2], Dq[3], Dq[4], Dq[5], Dq[6]}) :
>
> #Backward recursion: Link 5
> f5 := MatrixVectorMultiply(R56, f6) :
> τ5 := MatrixVectorMultiply(R56, τ6) + MatrixVectorMultiply(I5, α5) + CrossProduct(ω5,
     MatrixVectorMultiply(I5, ω5)) :
> τ5y := collect(combine(MatrixVectorMultiply(Transpose(b5), τ5), trig), {DDq[1], DDq[2], DDq[3], DDq[4],
     DDq[5], DDq[6], Dq[1], Dq[2], Dq[3], Dq[4], Dq[5], Dq[6]}) :
>
> #Backward recursion: Link 4
> f4 := MatrixVectorMultiply(R45, f5) + Add(m4·ac4,-m4·g4) :
> τ4 := MatrixVectorMultiply(R45, τ5) -CrossProduct(f4, r3c4) + CrossProduct(MatrixVectorMultiply(R45, f5),
     r4c4) + MatrixVectorMultiply(I4, α4) + CrossProduct(ω4, MatrixVectorMultiply(I4, ω4)) :
> τ4x := collect(combine(MatrixVectorMultiply(Transpose(b4), τ4), trig), {DDq[1], DDq[2], DDq[3], DDq[4],
     DDq[5], DDq[6], Dq[1], Dq[2], Dq[3], Dq[4], Dq[5], Dq[6]}) :
>
```

```
> #Backward recursion: Link 3
> f3 := MatrixVectorMultiply(R34, f4) :
> τ3 := MatrixVectorMultiply(R34, τ4) + MatrixVectorMultiply(I3, α3) + CrossProduct(ω3,
      MatrixVectorMultiply(I3, ω3)) :
> τ3y := collect(combine(MatrixVectorMultiply(Transpose(b3), τ3), trig), {DDq[1], DDq[2], DDq[3], DDq[4],
      DDq[5], DDq[6], Dq[1], Dq[2], Dq[3], Dq[4], Dq[5], Dq[6]}) :
>
> #Backward recursion: Link 2
> f2 := MatrixVectorMultiply(R23, f3) + Add(m2·ac2,-m2·g2) :
> τ2 := MatrixVectorMultiply(R23, τ3) - CrossProduct(f2, r1c2) + CrossProduct(MatrixVectorMultiply(R23, f3),
      r2c2) + MatrixVectorMultiply(I2, α2) + CrossProduct(ω2, MatrixVectorMultiply(I2, ω2)) :
> τ2z := collect(combine(MatrixVectorMultiply(Transpose(b2), τ2), trig), {DDq[1], DDq[2], DDq[3], DDq[4],
      DDq[5], DDq[6], Dq[1], Dq[2], Dq[3], Dq[4], Dq[5], Dq[6]}) :
>
> #Backward recursion: Link 1
> f1 := MatrixVectorMultiply(R12, f2) + Add(m1·ac1,-m1·g1) :
> τ1 := MatrixVectorMultiply(R12, τ2) - CrossProduct(f1, r0c1) + CrossProduct(MatrixVectorMultiply(R12, f2),
      r1c1) + MatrixVectorMultiply(I1, α1) + CrossProduct(ω1, MatrixVectorMultiply(I1, ω1)) :
> τ1x := collect(combine(MatrixVectorMultiply(Transpose(b1), τ1), trig), {DDq[1], DDq[2], DDq[3], DDq[4],
      DDq[5], DDq[6], Dq[1], Dq[2], Dq[3], Dq[4], Dq[5], Dq[6]}) :
>
> #Setting up the matrix elements
> m11 := eval(τ1x, {DDq[1] = 1, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m12 := eval(τ1x, {DDq[1] = 0, DDq[2] = 1, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m13 := eval(τ1x, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 1, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m14 := eval(τ1x, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 1, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m15 := eval(τ1x, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 1, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m16 := eval(τ1x, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 1, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
>
> m21 := eval(τ2z, {DDq[1] = 1, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m22 := eval(τ2z, {DDq[1] = 0, DDq[2] = 1, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m23 := eval(τ2z, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 1, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m24 := eval(τ2z, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 1, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m25 := eval(τ2z, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 1, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m26 := eval(τ2z, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 1, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
>
> m31 := eval(τ3y, {DDq[1] = 1, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m32 := eval(τ3y, {DDq[1] = 0, DDq[2] = 1, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m33 := eval(τ3y, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 1, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m34 := eval(τ3y, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 1, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m35 := eval(τ3y, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 1, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m36 := eval(τ3y, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 1, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
>
> m41 := eval(τ4x, {DDq[1] = 1, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m42 := eval(τ4x, {DDq[1] = 0, DDq[2] = 1, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m43 := eval(τ4x, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 1, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m44 := eval(τ4x, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 1, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m45 := eval(τ4x, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 1, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m46 := eval(τ4x, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 1, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
>
> m51 := eval(τ5y, {DDq[1] = 1, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m52 := eval(τ5y, {DDq[1] = 0, DDq[2] = 1, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m53 := eval(τ5y, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 1, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m54 := eval(τ5y, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 1, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m55 := eval(τ5y, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 1, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m56 := eval(τ5y, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 1, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
>
> m61 := eval(τ6z, {DDq[1] = 1, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m62 := eval(τ6z, {DDq[1] = 0, DDq[2] = 1, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m63 := eval(τ6z, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 1, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m64 := eval(τ6z, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 1, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
> m65 := eval(τ6z, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 1, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
```

```
> m66 := eval(τ6z, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 1, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0, g = 0}) :
>
> g6 := eval(τ6z, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0}) :
> g5 := eval(τ5y, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0}) :
> g4 := eval(τ4x, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0}) :
> g3 := eval(τ3y, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0}) :
> g2 := eval(τ2z, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0}) :
> g1 := eval(τ1x, {DDq[1] = 0, DDq[2] = 0, DDq[3] = 0, DDq[4] = 0, DDq[5] = 0, DDq[6] = 0, Dq[1] = 0,
      Dq[2] = 0, Dq[3] = 0, Dq[4] = 0, Dq[5] = 0, Dq[6] = 0}) :
>
> #Setting up the dynamic system
> M0 := Matrix([[m11, m12, m13, m14, m15, m16], [m21, m22, m23, m24, m25, m26], [m31, m32, m33, m34, m35,
      m36], [m41, m42, m43, m44, m45, m46], [m51, m52, m53, m54, m55, m56], [m61, m62, m63, m64, m65,
      m66]]) :
> G0 := Vector([[g1], [g2], [g3], [g4], [g5], [g6]]) :
> τ0 := Vector([[τ1x], [τ2z], [τ3y], [τ4x], [τ5y], [τ6z]]) :
> C0Dq := combine(τ0 − MatrixVectorMultiply(M0, DDq) − G0, trig) :
>
> #Kinetic energy
> Ekin := combine(1/2 · VectorMatrixMultiply(Transpose(Dq), MatrixVectorMultiply(M0, Dq)), trig) :
>
> #Matlab conversion
> with(CodeGeneration) :
> writeto("Ur5Dynamics.m");
>
> Matlab(m11, resultname = "m11");
> Matlab(m12, resultname = "m12");
> Matlab(m13, resultname = "m13");
> Matlab(m14, resultname = "m14");
> Matlab(m15, resultname = "m15");
> Matlab(m16, resultname = "m16");
>
> Matlab(m21, resultname = "m21");
> Matlab(m22, resultname = "m22");
> Matlab(m23, resultname = "m23");
> Matlab(m24, resultname = "m24");
> Matlab(m25, resultname = "m25");
> Matlab(m26, resultname = "m26");
>
> Matlab(m31, resultname = "m31");
> Matlab(m32, resultname = "m32");
> Matlab(m33, resultname = "m33");
> Matlab(m34, resultname = "m34");
> Matlab(m35, resultname = "m35");
> Matlab(m36, resultname = "m36");
>
> Matlab(m41, resultname = "m41");
> Matlab(m42, resultname = "m42");
> Matlab(m43, resultname = "m43");
> Matlab(m44, resultname = "m44");
> Matlab(m45, resultname = "m45");
> Matlab(m46, resultname = "m46");
>
> Matlab(m51, resultname = "m51");
> Matlab(m52, resultname = "m52");
> Matlab(m53, resultname = "m53");
> Matlab(m54, resultname = "m54");
> Matlab(m55, resultname = "m55");
> Matlab(m56, resultname = "m56");
>
> Matlab(m61, resultname = "m61");
> Matlab(m62, resultname = "m62");
> Matlab(m63, resultname = "m63");
> Matlab(m64, resultname = "m64");
> Matlab(m65, resultname = "m65");
> Matlab(m66, resultname = "m66");
>
> Matlab(g1, resultname = "g1");
> Matlab(g2, resultname = "g2");
> Matlab(g3, resultname = "g3");
> Matlab(g4, resultname = "g4");
> Matlab(g5, resultname = "g5");
> Matlab(g6, resultname = "g6");
>
> Matlab(C0Dq(1), resultname = "cdq1");
> Matlab(C0Dq(2), resultname = "cdq2");
> Matlab(C0Dq(3), resultname = "cdq3");
> Matlab(C0Dq(4), resultname = "cdq4");
> Matlab(C0Dq(5), resultname = "cdq5");
> Matlab(C0Dq(6), resultname = "cdq6");
>
> writeto(terminal);
> Matlab(Ekin, resultname = "K")
>
```

B.2. Dynamics (Matlab file)

The resulting dynamics are generated from the framework in the previous section and slightly modified for compatibility with the Matlab syntax.

```
%The expression for the elements of M,C,g are copied from Maple, and state ...
   derivatives have been redefined
m11 = cos(q5) * 0.29173632000000014e-2 + cos((2 * q2)) * 0.41364213499999911e0 + ...
   cos((2 * q2) + 0.2e1 * q5) * (-0.162630325872825665e-18) + cos((2 * q2) - 0.2e1 ...
   * q5) * (-0.162630325872825665e-18) + cos((2 * q4)) * ...
   (-0.433680868994201774e-18) + cos((2 * q2 + 2 * q4)) * 0.433680868994201774e-18 ...
   + cos((2 * q3 + 2 * q4)) * 0.433680868994201774e-18 + cos(0.2e1 * q5 + (2 * q4) ...
   + (2 * q3) + (2 * q2)) * (-0.132212075305000016e-3) + cos(-0.2e1 * q5 + (2 * q4) ...
   + (2 * q3) + (2 * q2)) * (-0.132212075305000016e-3) + cos(0.2e1 * q5) * ...
   0.264424150609999706e-3 + cos((2 * q4 + 2 * q3 + 2 * q2)) * ...
   (-0.462956163999961659e-5) + cos(q5 + (2 * q4) + (2 * q3) + (2 * q2)) * ...
   0.813151629364128326e-19 + cos(-q5 + (2 * q4) + (2 * q3) + (2 * q2)) * ...
   0.813151629364128326e-19 + sin((q4 + q3 + 2 * q2)) * (-0.777807799999999990e-2) ...
   + sin((q4 + q3)) * (-0.777807799999999990e-2) + sin(q5 + q4 + q3 + (2 * q2)) * ...
   0.284376000000000030e-2 + sin(-q5 + q4 + q3 + (2 * q2)) * ...
   (-0.284376000000000030e-2) + sin(q5 + q4 + q3) * 0.284376000000000030e-2 + ...
   sin(-q5 + q4 + q3) * (-0.284376000000000030e-2) + 0.578636171160119894e0 + ...
   cos((2 * q2) + (2 * q4) - q5) * (-0.271050543121376109e-19) + cos((2 * q3) + (2 ...
   * q4) + q5) * 0.271050543121376109e-19 + cos((2 * q4) + q5) * ...
   0.271050543121376109e-19 + cos((2 * q2) + (2 * q4) + q5) * ...
   (-0.271050543121376109e-19) + cos((2 * q3 + 2 * q2)) * 0.225799867760000036e-1 + ...
   cos((2 * q3) + (2 * q4) - q5) * 0.271050543121376109e-19 + cos((2 * q4) - q5) * ...
   0.271050543121376109e-19;
m12 = cos((q2 - 2 * q5 - q4 - q3)) * 0.325260651745651330e-18 + cos((2 * q5 + q4 + ...
   q3 + q2)) * (-0.264424150609999706e-3) + cos((q5 + q4 + q3 + q2)) * ...
   (-0.729340800000000036e-3) + sin((q5 + q2)) * 0.284376000000000030e-2 + sin((-q5 ...
   + q2)) * 0.284376000000000030e-2 + cos((-2 * q5 + q4 + q3 + q2)) * ...
   0.264424150610000032e-3 + cos((-q5 + q4 + q3 + q2)) * 0.729340800000000036e-3 + ...
   sin(q2) * 0.267177490999999989e0 + cos((q4 + q3 + q2)) * 0.202230028000000006e-2;
m13 = cos(q4 + q3 + q2) * 0.202230028000000006e-2 + cos(q5 + q4 + q3 + q2) * ...
   (-0.729340800000000036e-3) + cos(-0.2e1 * q5 + q4 + q3 + q2) * ...
   0.264424150610000032e-3 + cos(-q5 + q4 + q3 + q2) * 0.729340800000000036e-3 + ...
   cos(0.2e1 * q5 + q4 + q3 + q2) * (-0.264424150610000032e-3);
m14 = cos(q4 + q3 + q2) * 0.202230028000000006e-2 + cos(q5 + q4 + q3 + q2) * ...
   (-0.729340800000000036e-3) + cos(-0.2e1 * q5 + q4 + q3 + q2) * ...
   0.264424150610000032e-3 + cos(-q5 + q4 + q3 + q2) * 0.729340800000000036e-3 + ...
   cos(0.2e1 * q5 + q4 + q3 + q2) * (-0.264424150610000032e-3);
m15 = sin(-q5 + q2) * 0.284376000000000030e-2 + cos(q4 + q3 + q2) * ...
   (-0.201419266993000008e-2) + cos(-q5 + q4 + q3 + q2) * ...
   (-0.729340800000000036e-3) + cos(q5 + q4 + q3 + q2) * (-0.729340800000000036e-3) ...
   + sin(q5 + q2) * (-0.284376000000000030e-2);
m16 = -0.6087959740e-4 * cos(-q5 + q4 + q3 + q2) + 0.6087959740e-4 * cos(q5 + q4 + ...
   q3 + q2);
m21 = sin(q2) * 0.267177490999999989e0 + cos(q4 + q3 + q2) * ...
   0.202230028000000006e-2 + cos(-(2 * q5) + q4 + q3 + q2) * ...
   0.264424150610000032e-3 + sin(q5 + q2) * 0.284376000000000030e-2 + sin(-q5 + q2) ...
   * 0.284376000000000030e-2 + cos((2 * q5) + q4 + q3 + q2) * ...
   (-0.264424150610000032e-3) + cos(-q5 + q4 + q3 + q2) * 0.729340800000000036e-3 + ...
   cos(q5 + q4 + q3 + q2) * (-0.729340800000000036e-3);
m22 = sin(-q5 + q4 + q3) * (-0.568752000000000060e-2) + sin(q4 + q3) * ...
   (-0.155561560000000015e-1) + 0.885518424388219927e0 + cos(0.2e1 * q5) * ...
   (-0.528848301220000063e-3) + sin(q5 + q4 + q3) * 0.568752000000000060e-2;
m23 = cos((2 * q5)) * (-0.528848301220000063e-3) + sin((-q5 + q4 + q3)) * ...
   (-0.284376000000000030e-2) + sin((q5 + q4 + q3)) * 0.284376000000000030e-2 + ...
   0.496708343882199949e-1 + sin((q4 + q3)) * (-0.777807799999999903e-2);
m24 = sin(q5 + q4 + q3) * 0.284376000000000030e-2 + cos(0.2e1 * q5) * ...
   (-0.528848301220000063e-3) + sin(q4 + q3) * (-0.777807799999999903e-2) + sin(-q5 ...
   + q4 + q3) * (-0.284376000000000030e-2) + 0.291008553822000044e-2;
m25 = sin(-q5 + q4 + q3) * 0.284376000000000030e-2 + sin(q5 + q4 + q3) * ...
   0.284376000000000030e-2;
m26 = 0.1217591948e-3 * cos(q5);
m31 = cos(q4 + q3 + q2) * 0.202230028000000006e-2 + cos(q5 + q4 + q3 + q2) * ...
   (-0.729340800000000036e-3) + cos(-0.2e1 * q5 + q4 + q3 + q2) * ...
   0.264424150610000032e-3 + cos(-q5 + q4 + q3 + q2) * 0.729340800000000036e-3 + ...
   cos(0.2e1 * q5 + q4 + q3 + q2) * (-0.264424150610000032e-3);
```

```
m32 = sin(−q5 + q4 + q3) * (−0.284376000000000030e−2) + cos(0.2e1 * q5) * ...
   (−0.528848301220000063e−3) + sin(q4 + q3) * (−0.777807800000000077e−2) + ...
   0.496708343882199949e−1 + sin(q5 + q4 + q3) * 0.284376000000000030e−2;
m33 = 0.496708343882199949e−1 + cos((2 * q5)) * (−0.528848301220000063e−3);
m34 = 0.291008553822000044e−2 + cos((2 * q5)) * (−0.528848301220000063e−3);
m35 = 0.0e0;
m36 = 0.1217591948e−3 * cos(q5);
m41 = cos(q4 + q3 + q2) * 0.202230028000000006e−2 + cos(q5 + q4 + q3 + q2) * ...
   (−0.729340800000000036e−3) + cos(−0.2e1 * q5 + q4 + q3 + q2) * ...
   0.264424150610000032e−3 + cos(−q5 + q4 + q3 + q2) * 0.729340800000000036e−3 + ...
   cos(0.2e1 * q5 + q4 + q3 + q2) * (−0.264424150610000032e−3);
m42 = sin(−q5 + q4 + q3) * (−0.284376000000000030e−2) + sin(q5 + q4 + q3) * ...
   0.284376000000000030e−2 + cos(0.2e1 * q5) * (−0.528848301220000063e−3) + sin(q4 ...
   + q3) * (−0.777807800000000077e−2) + 0.291008553822000044e−2;
m43 = 0.291008553822000001e−2 + cos((2 * q5)) * (−0.528848301220000063e−3);
m44 = 0.291008553822000044e−2 + cos((2 * q5)) * (−0.528848301220000063e−3);
m45 = 0.0e0;
m46 = 0.1217591948e−3 * cos(q5);
m51 = sin(−q5 + q2) * 0.284376000000000030e−2 + cos(q5 + q4 + q3 + q2) * ...
   (−0.729340800000000036e−3) + sin(q5 + q2) * (−0.284376000000000030e−2) + cos(q4 ...
   + q3 + q2) * (−0.201419266993999990e−2) + cos(−q5 + q4 + q3 + q2) * ...
   (−0.729340800000000036e−3);
m52 = sin(−q5 + q4 + q3) * 0.284376000000000030e−2 + sin(q5 + q4 + q3) * ...
   0.284376000000000030e−2;
m53 = 0.0e0;
m54 = 0.0e0;
m55 = 0.201419266993000008e−2;
m56 = 0.0e0;
m61 = −0.6087959740e−4 * cos(−q5 + q4 + q3 + q2) + 0.6087959740e−4 * cos(q5 + q4 + ...
   q3 + q2);
m62 = 0.1217591948e−3 * cos(q5);
m63 = 0.1217591948e−3 * cos(q5);
m64 = 0.1217591948e−3 * cos(q5);
m65 = 0.0e0;
m66 = 0.1217591948e−3;
g1 = 0;
g2 = g * sin(−q5 + q4 + q3 + q2) * 0.669120000000000045e−2 + cos(q2) * g * ...
   (−0.237166999999999994e1) + g * sin(q5 + q4 + q3 + q2) * ...
   (−0.669120000000000045e−2) + sin(q4 + q3 + q2) * g * 0.183013599999999992e−1;
g3 = g * sin(q5 + q4 + q3 + q2) * (−0.669120000000000045e−2) + sin(q4 + q3 + q2) * ...
   g * 0.183013599999999992e−1 + g * sin(−q5 + q4 + q3 + q2) * 0.669120000000000045e−2;
g4 = g * sin(q5 + q4 + q3 + q2) * (−0.669120000000000045e−2) + sin(q4 + q3 + q2) * ...
   g * 0.183013599999999992e−1 + g * sin(−q5 + q4 + q3 + q2) * 0.669120000000000045e−2;
g5 = g * sin(−q5 + q4 + q3 + q2) * (−0.669120000000000045e−2) + g * sin(q5 + q4 + ...
   q3 + q2) * (−0.669120000000000045e−2);
g6 = 0.0e0;
cdq1 = 0.5000000000e−14 * Dq3 * Dq6 * sin((2 * q6 + q4 + q3 + q2 − q5)) + ...
   0.2500000000e−14 * Dq3 * Dq6 * sin((−2 * q6 + q4 + q3 + q2 + 2 * q5)) − ...
   0.2500000000e−14 * Dq1 * Dq6 * sin((−2 * q6 + 2 * q4 + 2 * q3 + 2 * q2 − q5)) − ...
   0.2500000000e−14 * Dq1 * Dq6 * sin((2 * q6 + q5)) − 0.2500000000e−14 * Dq1 * Dq6 ...
   * sin((−2 * q6 + 2 * q4 + 2 * q3 + 2 * q2 + q5)) − 0.5000000000e−14 * Dq1 * Dq6 ...
   * sin((2 * q4 + 2 * q3 + 2 * q2)) + 0.5000000000e−14 * Dq1 * Dq6 * sin((−2 * q6 ...
   + 2 * q4 + 2 * q3 + 2 * q2)) + Dq1 * Dq2 * cos((q5 + q4 + q3 + 2 * q2)) * ...
   0.568752000000000060e−2 + Dq1 * Dq2 * cos((−q5 + q4 + q3 + 2 * q2)) * ...
   (−0.568752000000000060e−2) + Dq1 * Dq2 * cos((q5 + q4 + q3)) * ...
   (−0.433680868994201774e−18) + Dq1 * Dq2 * sin((2 * q2 + 2 * q5)) * ...
   0.499999889581881352e−13 + Dq1 * Dq2 * sin((2 * q2)) * (−0.827284270000549826e0) ...
   + Dq1 * Dq2 * sin((2 * q5 + 2 * q4 + 2 * q3 + 2 * q2)) * 0.264424150710000009e−3 ...
   + Dq1 * Dq2 * sin((2 * q3)) * 0.499999347480795109e−12 + Dq1 * Dq2 * sin((2 * q4 ...
   + 2 * q3 + 2 * q2)) * 0.925912333000003529e−5 + Dq1 * Dq2 * sin((2 * q3 + 2 * ...
   q4)) * 0.499999347480795109e−13 + Dq1 * Dq2 * sin((2 * q2 − 2 * q5)) * ...
   (−0.499996908025907016e−13) + Dq1 * Dq2 * sin((2 * q3 + 2 * q2)) * ...
   (−0.451599735515000045e−1) + Dq1 * Dq2 * sin((2 * q3 + 2 * q4 − 2 * q5)) * ...
   0.499999889581881352e−13 + Dq1 * Dq2 * sin((2 * q3 + 2 * q4 + 2 * q5)) * ...
   (−0.499999889581881352e−13) + Dq1 * Dq2 * cos((−q5 + q4 + q3)) * ...
   0.433680868994201774e−18 + Dq1 * Dq2 * sin((−2 * q5 + 2 * q4 + 2 * q3 + 2 * q2)) ...
   * 0.264424150710000009e−3 + Dq1 * Dq2 * cos((q4 + q3)) * ...
   0.173472347597680709e−17 + Dq1 * Dq2 * cos((q4 + q3 + 2 * q2)) * ...
   (−0.155561559999999998e−1) + Dq1 * Dq3 * sin((2 * q3 + 2 * q4)) * ...
   (−0.499998805379708866e−13) + Dq1 * Dq3 * sin((2 * q2 − 2 * q5)) * ...
   0.499999889581881352e−13 + Dq1 * Dq3 * sin((2 * q3 + 2 * q4 − 2 * q5)) * ...
   (−0.499999889581881352e−13) + Dq1 * Dq3 * sin((2 * q2 + 2 * q4 + 2 * q5)) * ...
   0.135525271560688054e−19 + Dq1 * Dq3 * sin((2 * q4 + 2 * q5)) * ...
   (−0.135525271560688054e−19) + Dq1 * Dq3 * sin((2 * q4 − 2 * q5))...
```

```
* (-0.135525271560688054e-19) + Dq1 * Dq3 * sin((2 * q2 + 2 * q4 - 2 * q5)) * ...
   0.135525271560688054e-19 + Dq1 * Dq3 * sin((2 * q2 + 2 * q3 - 2 * q5)) * ...
   (-0.135525271560688054e-19) + Dq1 * Dq3 * sin((2 * q3 - 2 * q5)) * ...
   0.135525271560688054e-19 + Dq1 * Dq3 * sin((2 * q3 + 2 * q5)) * ...
   0.135525271560688054e-19 + Dq1 * Dq3 * sin((2 * q2 + 2 * q3 + 2 * q5)) * ...
   (-0.135525271560688054e-19) + Dq1 * Dq3 * sin((2 * q2 + 2 * q4)) * ...
   (-0.542101086242752217e-19) + Dq1 * Dq3 * sin((2 * q4)) * ...
   0.542101086242752217e-19 + Dq1 * Dq3 * sin((2 * q2)) * 0.500016694715554877e-13 ...
   + Dq1 * Dq3 * sin((2 * q3 + 2 * q2)) * (-0.451599735520000004e-1) + Dq1 * Dq3 * ...
   sin((2 * q4 + 2 * q3 + 2 * q2)) * 0.925912332999998108e-5 + Dq1 * Dq3 * cos((q4 ...
   + q3 + 2 * q2)) * (-0.777807799999999903e-2) + Dq1 * Dq3 * cos((-q5 + q4 + q3 + ...
   2 * q2)) * (-0.284376000000000030e-2) + Dq1 * Dq3 * sin((-2 * q5 + 2 * q4 + 2 * ...
   q3 + 2 * q2)) * 0.264424150709999955e-3 + Dq1 * Dq3 * sin((2 * q5 + 2 * q4 + 2 * ...
   q3 + 2 * q2)) * 0.264424150659999966e-3 + Dq1 * Dq3 * cos((q5 + q4 + q3 + 2 * ...
   q2)) * 0.284376000000000030e-2 + Dq1 * Dq4 * cos((-q5 + q4 + q3 + 2 * q2)) * ...
   (-0.284376000000000030e-2) - 0.6087959740e-4 * Dq4 * Dq6 * sin((q5 + q4 + q3 + ...
   q2)) - 0.6087959740e-4 * Dq6 * Dq5 * sin((-q5 + q4 + q3 + q2)) - 0.6087959740e-4 ...
   * Dq6 * Dq5 * sin((q5 + q4 + q3 + q2)) + Dq1 * Dq4 * sin((-2 * q5 + 2 * q4 + 2 * ...
   q3 + 2 * q2)) * 0.264424150784999966e-3 + Dq1 * Dq4 * sin((2 * q3 + 2 * q4 - 2 * ...
   q5)) * (-0.124999999500524650e-12) + Dq1 * Dq4 * sin((2 * q5 + 2 * q4 + 2 * q3 + ...
   2 * q2)) * 0.264424150709999955e-3 + Dq1 * Dq4 * sin((2 * q2 + 2 * q3 - 2 * q5)) ...
   * (-0.135525271560688054e-19) + Dq1 * Dq4 * sin((2 * q3 - 2 * q5)) * ...
   0.135525271560688054e-19 + Dq1 * Dq4 * sin((2 * q4 + 2 * q3 + 2 * q2)) * ...
   0.925912357999992587e-5 + Dq1 * Dq4 * sin((2 * q3 + 2 * q4)) * ...
   (-0.199999955832752541e-12) + Dq1 * Dq4 * sin((2 * q2)) * ...
   0.199999955832752541e-12 + Dq1 * Dq4 * cos((q5 + q4 + q3 + 2 * q2)) * ...
   0.284376000000000030e-2 + Dq1 * Dq4 * cos((q4 + q3 + 2 * q2)) ...
* (-0.777807799999999903e-2) + Dq1 * Dq4 * sin((2 * q3 + 2 * q4 + 2 * q5)) * ...
   0.499999889581881352e-13 + Dq1 * Dq4 * sin((2 * q2 + 2 * q5)) * ...
   (-0.499999889581881352e-13) + Dq1 * Dq4 * sin((2 * q2 - 2 * q5)) * ...
   0.124999999500524650e-12 + Dq1 * Dq4 * sin((2 * q2 + 2 * q4 - 2 * q5)) * ...
   0.135525271560688054e-19 + Dq1 * Dq4 * sin((2 * q4 - 2 * q5)) * ...
   (-0.135525271560688054e-19) + Dq1 * Dq5 * sin((2 * q5 + 2 * q4 + 2 * q3 + 2 * ...
   q2)) * 0.264424150660000021e-3 + Dq1 * Dq5 * sin((-2 * q5 + 2 * q4 + 2 * q3 + 2 ...
   * q2)) * (-0.264424150660000021e-3) + Dq1 * Dq5 * cos((-q5 + q4 + q3 + 2 * q2)) ...
   * 0.284376000000000030e-2 + Dq1 * Dq5 * cos((q5 + q4 + q3 + 2 * q2)) * ...
   0.284376000000000030e-2 + Dq1 * Dq5 * sin(q5) * (-0.291736320000000014e-2) + Dq1 ...
   * Dq3 * cos((q4 + q3)) * (-0.777807799999999903e-2) + Dq1 * Dq3 * cos((q5 + q4 + ...
   q3)) * 0.284376000000000030e-2 + Dq1 * Dq3 * cos((-q5 + q4 + q3)) * ...
   (-0.284376000000000030e-2) + Dq1 * Dq4 * cos((q4 + q3)) * ...
   (-0.777807799999999903e-2) + Dq1 * Dq4 * cos((q5 + q4 + q3)) * ...
   0.284376000000000030e-2 + Dq1 * Dq4 * cos((-q5 + q4 + q3)) * ...
   (-0.284376000000000030e-2) + Dq1 * Dq5 * cos((q5 + q4 + q3)) * ...
   0.284376000000000030e-2 + Dq1 * Dq5 * cos((-q5 + q4 + q3)) * ...
   0.284376000000000030e-2 + Dq1 * sin((2 * q5)) * Dq2 * ...
   (-0.298155597433513719e-18) + Dq1 * Dq5 * sin((2 * q5)) * ...
   (-0.528848301320000041e-3) - 0.5000000000e-14 * Dq1 * Dq6 * sin((2 * q6)) + Dq1 ...
   * Dq3 * sin((2 * q5)) * 0.499999889581881352e-13 + Dq1 * Dq4 * sin((2 * q5)) * ...
   0.750000105423365149e-13 + 0.2500000000e-14 * Dq1 * Dq6 * sin((-2 * q6 + q5)) + ...
   Dq2 * Dq3 * sin((2 * q5 + q4 + q3 + q2)) * 0.528848301319999933e-3 + Dq2 * Dq3 * ...
   sin((q2 - q3 + q4 + 2 * q5)) * 0.271050543121376109e-19 + Dq2 * Dq3 * sin((q2 + ...
   q3 - q4 + 2 * q5)) * (-0.271050543121376109e-19) + Dq2 * Dq3 * sin((q4 + q3 + ...
   q2))...
* (-0.404460056000000012e-2) + Dq2 * Dq3 * sin((-q5 + q4 + q3 + q2)) * ...
   (-0.145868160000000007e-2) + Dq2 * Dq3 * sin((q5 + q4 + q3 + q2)) * ...
   0.145868160000000007e-2 + Dq2 * Dq3 * sin((q2 - q3 + q4 - 2 * q5)) * ...
   (-0.271050543121376109e-19) + Dq2 * Dq3 * sin((q2 + q3 - q4 - 2 * q5)) * ...
   0.271050543121376109e-19 + Dq2 * Dq3 * sin((-2 * q5 + q4 + q3 + q2)) * ...
   (-0.528848301319999933e-3) + Dq2 * Dq4 * sin((q5 + q4 + q3 + q2)) * ...
   0.145868160000000007e-2 + Dq2 * Dq4 * sin((-q5 + q4 + q3 + q2)) * ...
   (-0.145868160000000007e-2) + Dq2 * Dq4 * sin((-2 * q5 + q4 + q3 + q2)) * ...
   (-0.528848301320000041e-3) + Dq2 * Dq4 * sin((2 * q5 + q4 + q3 + q2)) * ...
   0.528848301320000041e-3 + Dq2 * Dq4 * sin((q4 + q3 + q2)) * ...
   (-0.404460056000000012e-2) + Dq2 * Dq5 * sin((-2 * q5 + q4 + q3 + q2)) * ...
   0.528848301245000030e-3 + Dq2 * Dq5 * sin((q5 + q4 + q3 + q2)) * ...
   0.145868160000000007e-2 + Dq2 * Dq5 * sin((-q5 + q4 + q3 + q2)) * ...
   0.145868160000000007e-2 + Dq2 * Dq5 * sin((2 * q5 + q4 + q3 + q2)) * ...
   0.528848301245000030e-3 + Dq2 * Dq5 * sin((q4 + q3 + q2)) * ...
   0.201419267009000013e-2 - 0.5000000000e-14 * Dq2 * Dq6 * sin((-2 * q5 + q4 + q3 ...
   + q2)) - 0.6087959740e-4 * Dq2 * Dq6 * sin((q5 + q4 + q3 + q2)) + ...
   0.6087959740e-4 * Dq2 * Dq6 * sin((-q5 + q4 + q3 + q2)) - 0.5000000000e-14 * Dq2 ...
   * Dq6 * sin((2 * q5 + q4 + q3 + q2)) - 0.1000000000e-13 * Dq2 * Dq6 * sin((q4 + ...
   q3 + q2)) + 0.2500000000e-14 * Dq3 * Dq6 * sin((-2 * q5 + q4 + q3 + q2)) + ...
   0.2500000000e-14 * Dq3 * Dq6 * sin((2 * q5 + q4 + q3 + q2)) + Dq3 * Dq4 * ...
```

```
    sin((q2 + q3 - q4 - 2 * q5)) * (-0.271050543121376109e-19) + Dq3 * Dq4 * sin((q2 ...
    - q3 + q4 + 2 * q5)) * (-0.271050543121376109e-19) + Dq3 * Dq4 * sin((q2 - 2 * ...
    q5 - q4 - q3)) * (-0.542101086242752217e-19) + Dq3 * Dq4 * sin((q2 + q3 - q4 + 2 ...
    * q5)) ...
 * 0.271050543121376109e-19 + Dq3 * Dq4 * sin((q2 - q3 + q4 - 2 * q5)) * ...
    0.271050543121376109e-19 + Dq3 * Dq4 * sin((q2 + 2 * q5 - q4 - q3)) * ...
    0.542101086242752217e-19 + Dq3 * Dq5 * sin((q2 + 2 * q5 - q4 - q3)) * ...
    0.625000268553166372e-13 + Dq3 * Dq5 * sin((q2 - 2 * q5 - q4 - q3)) * ...
    0.625000268553166372e-13 + Dq3 * Dq5 * sin((-q3 - q4 + q2)) * ...
    0.125000053710633274e-12 + 0.2500000000e-14 * Dq3 * Dq6 * sin((2 * q6 + q4 + q3 ...
    + q2 - 2 * q5)) + 0.2500000000e-14 * Dq3 * Dq6 * sin((2 * q6 + q4 + q3 + q2)) + ...
    Dq3 * Dq5 * sin((q4 + q3 + q2)) * 0.201419267021500040e-2 + Dq3 * Dq5 * sin((2 * ...
    q5 + q4 + q3 + q2)) * 0.528848301307500057e-3 + Dq3 * Dq5 * sin((-2 * q5 + q4 + ...
    q3 + q2)) * 0.528848301307500057e-3 - 0.5000000000e-14 * Dq3 * Dq6 * sin((-2 * ...
    q6 + q4 + q3 + q2 + q5)) + Dq3 * Dq4 * sin((2 * q5 + q4 + q3 + q2)) * ...
    0.528848301319999933e-3 + Dq3 * Dq4 * sin((-2 * q5 + q4 + q3 + q2)) * ...
    (-0.528848301319999933e-3) + Dq3 * Dq4 * sin((q4 + q3 + q2)) * ...
    (-0.404460056000000012e-2) + 0.6087959740e-4 * Dq3 * Dq6 * sin((-q5 + q4 + q3 + ...
    q2)) - 0.6087959740e-4 * Dq3 * Dq6 * sin((q5 + q4 + q3 + q2)) + 0.5000000000e-14 ...
    * Dq3 * Dq6 * sin((q4 + q3 + q2)) + 0.2500000000e-14 * Dq3 * Dq6 * sin((-2 * q6 ...
    + q4 + q3 + q2)) + Dq3 * Dq4 * sin((q5 + q4 + q3 + q2)) * ...
    0.145868160000000007e-2 + Dq3 * Dq4 * sin((-q5 + q4 + q3 + q2)) * ...
    (-0.145868160000000007e-2) + Dq3 * Dq5 * sin((-q5 + q4 + q3 + q2)) * ...
    0.145868160000000007e-2 + Dq3 * Dq5 * sin((q5 + q4 + q3 + q2)) * ...
    0.145868160000000007e-2 + Dq4 * Dq5 * sin((-q5 + q4 + q3 + q2)) * ...
    0.145868160000000007e-2 + Dq4 * Dq5 * sin((q2 - 2 * q5 - q4 - q3)) * ...
    0.750000105423365149e-13 + Dq4 * Dq5 * sin((q4 + q3 + q2)) * ...
    0.201419266993999990e-2 + Dq4 * Dq5 * sin((2 * q5 + q4 + q3 + q2)) * ...
    0.528848301320000041e-3 + Dq4 * Dq5 * sin((q2 + 2 * q5 - q4 - q3))...
 * 0.750000105423365149e-13 + Dq4 * Dq5 * sin((q5 + q4 + q3 + q2)) * ...
    0.145868160000000007e-2 + Dq4 * Dq5 * sin((-q3 - q4 + q2)) * ...
    (-0.150000021084673030e-12) + Dq4 * Dq5 * sin((-2 * q5 + q4 + q3 + q2)) * ...
    0.528848301320000041e-3 - 0.2500000000e-14 * Dq4 * Dq6 * sin((-2 * q6 + q4 + q3 ...
    + q2 + 2 * q5)) - 0.2500000000e-14 * Dq4 * Dq6 * sin((2 * q6 + q4 + q3 + q2)) - ...
    0.2500000000e-14 * Dq4 * Dq6 * sin((-2 * q6 + q4 + q3 + q2)) - 0.2500000000e-14 ...
    * Dq4 * Dq6 * sin((2 * q6 + q4 + q3 + q2 - 2 * q5)) + 0.5000000000e-14 * Dq4 * ...
    Dq6 * sin((q4 + q3 + q2)) + 0.5000000000e-14 * Dq4 * Dq6 * sin((-2 * q6 + q4 + ...
    q3 + q2 + q5)) + 0.2500000000e-14 * Dq4 * Dq6 * sin((-2 * q5 + q4 + q3 + q2)) - ...
    0.5000000000e-14 * Dq4 * Dq6 * sin((2 * q6 + q4 + q3 + q2 - q5)) + ...
    0.2500000000e-14 * Dq4 * Dq6 * sin((2 * q5 + q4 + q3 + q2)) + 0.6087959740e-4 * ...
    Dq4 * Dq6 * sin((-q5 + q4 + q3 + q2)) + Dq4 ^ 2 * sin((q2 + 2 * q5 - q4 - q3)) * ...
    0.271050543121376109e-19 + Dq4 ^ 2 * sin((q2 + q3 - q4 - 2 * q5)) * ...
    (-0.135525271560688054e-19) + Dq4 ^ 2 * sin((q2 - q3 + q4 + 2 * q5)) * ...
    (-0.135525271560688054e-19) + Dq4 ^ 2 * sin((2 * q5 + q4 + q3 + q2)) * ...
    0.264424150659999966e-3 + Dq4 ^ 2 * sin((q2 - 2 * q5 - q4 - q3)) * ...
    (-0.271050543121376109e-19) + Dq4 ^ 2 * sin((q2 + q3 - q4 + 2 * q5)) * ...
    0.135525271560688054e-19 + Dq4 ^ 2 * sin((q2 - q3 + q4 - 2 * q5)) * ...
    0.135525271560688054e-19 + Dq4 ^ 2 * sin((-2 * q5 + q4 + q3 + q2)) * ...
    (-0.264424150659999966e-3) + Dq4 ^ 2 * sin((q4 + q3 + q2)) * ...
    (-0.202230028000000006e-2) + Dq4 ^ 2 * sin((q5 + q4 + q3 + q2)) * ...
    0.729340800000000036e-3 + Dq4 ^ 2 * sin((-q5 + q4 + q3 + q2)) * ...
    (-0.729340800000000036e-3) + Dq5 ^ 2 * sin((-q5 + q4 + q3 + q2)) * ...
    (-0.729340800000000036e-3) + Dq5 ^ 2 * cos((q5 + q2)) * ...
    (-0.284376000000000030e-2) + Dq5 ^ 2 * sin((q5 + q4 + q3 + q2)) ...
 * 0.729340800000000036e-3 + Dq5 ^ 2 * cos((-q5 + q2)) * (-0.284376000000000030e-2) ...
    + Dq3 ^ 2 * sin((q2 - q3 + q4 + 2 * q5)) * 0.135525271560688054e-19 + Dq3 ^ 2 * ...
    sin((-2 * q5 + q4 + q3 + q2)) * (-0.264424150659999966e-3) + Dq3 ^ 2 * sin((2 * ...
    q5 + q4 + q3 + q2)) * 0.264424150659999966e-3 + Dq3 ^ 2 * sin((q4 + q3 + q2)) * ...
    (-0.202230028000000006e-2) + Dq3 ^ 2 * sin((q2 + q3 - q4 - 2 * q5)) * ...
    0.135525271560688054e-19 + Dq3 ^ 2 * sin((q5 + q4 + q3 + q2)) * ...
    0.729340800000000036e-3 + Dq3 ^ 2 * sin((-q5 + q4 + q3 + q2)) * ...
    (-0.729340800000000036e-3) + Dq3 ^ 2 * sin((q2 + q3 - q4 + 2 * q5)) * ...
    (-0.135525271560688054e-19) + Dq3 ^ 2 * sin((q2 - q3 + q4 - 2 * q5)) * ...
    (-0.135525271560688054e-19) + Dq2 ^ 2 * sin((q2 + 2 * q5 - q4 - q3)) * ...
    0.271050543121376109e-19 + Dq2 ^ 2 * sin((-2 * q5 + q4 + q3 + q2)) * ...
    (-0.264424150659999966e-3) + Dq2 ^ 2 * sin((q4 + q3 + q2)) * ...
    (-0.202230028000000006e-2) + Dq2 ^ 2 * sin((q2 - 2 * q5 - q4 - q3)) * ...
    0.325260651745651330e-18 + Dq2 ^ 2 * cos((-q5 + q2)) * 0.284376000000000030e-2 + ...
    Dq2 ^ 2 * cos((q5 + q2)) * 0.284376000000000030e-2 + Dq2 ^ 2 * sin((2 * q5 + q4 ...
    + q3 + q2)) * 0.264424150659999641e-3 + Dq2 ^ 2 * sin((q2 + q3 - q4 - 2 * q5)) * ...
    (-0.135525271560688054e-19) + Dq2 ^ 2 * sin((q2 - q3 + q4 + 2 * q5)) * ...
    (-0.135525271560688054e-19) + Dq2 ^ 2 * sin((q2 + q3 - q4 + 2 * q5)) * ...
    0.135525271560688054e-19 + Dq2 ^ 2 * sin((q2 - q3 + q4 - 2 * q5)) * ...
    0.135525271560688054e-19 + Dq2 ^ 2 * cos(q2) * 0.267177490999999989e0 + Dq2 ^ 2 ...
```

```
   * sin((q5 + q4 + q3 + q2)) * 0.72934080000000036e-3 + Dq2 ^ 2 * sin((-q5 + q4 + ...
   q3 + q2)) * (-0.72934080000000036e-3) + Dq1 ^ 2 * sin((q2 + q3 - q4 + 2 * q5)) ...
   * (-0.338813178901720136e-19) + Dq1 ^ 2 * sin((q2 - q3 + q4 - 2 * q5)) * ...
   (-0.338813178901720136e-19) + Dq1 ^ 2 * sin((q2 - 2 * q5 - q4 - q3)) * ...
   0.243945488809238498e-18 + Dq1 ^ 2 * sin((q2 - q3 + q4 + 2 * q5)) * ...
   0.338813178901720136e-19 + Dq1 ^ 2 * sin((q2 + q3 - q4 - 2 * q5)) * ...
   0.338813178901720136e-19 + Dq1 ^ 2 * cos((2 * q3 + 2 * q4 - q5 + q2)) ...
 * 0.542101086242752217e-19 + Dq1 ^ 2 * sin((q2 + 2 * q5 - q4 - q3)) * ...
   (-0.271050543121376109e-19) + Dq1 ^ 2 * cos(q2) * 0.277555756156289135e-16 + Dq1 ...
   ^ 2 * sin((3 * q2 - 2 * q5 + q4 + q3)) * 0.948676900924816380e-19 + Dq1 ^ 2 * ...
   sin((3 * q2 + 2 * q5 + q4 + q3)) * 0.271050543121376109e-19 + Dq1 ^ 2 * sin((3 * ...
   q2 + 3 * q3 + 3 * q4 - q5)) * 0.271050543121376109e-19 + Dq1 ^ 2 * cos((3 * q2 + ...
   q5 + 2 * q4 + 2 * q3)) * 0.542101086242752217e-19 + Dq1 ^ 2 * cos((3 * q2 - q5 + ...
   2 * q4 + 2 * q3)) * 0.542101086242752217e-19 + Dq1 ^ 2 * cos((2 * q3 + 2 * q4 + ...
   q5 + q2)) * 0.542101086242752217e-19 + Dq1 ^ 2 * sin((3 * q2 + 3 * q3 + 3 * q4 + ...
   q5)) * (-0.271050543121376109e-19) + Dq1 ^ 2 * sin((3 * q2 + q3 + 3 * q4 + 2 * ...
   q5)) * (-0.338813178901720136e-20) + Dq1 ^ 2 * sin((q2 + q3 + 3 * q4 + 2 * q5)) ...
   * 0.338813178901720136e-20 + Dq1 ^ 2 * sin((3 * q2 + 3 * q3 + 3 * q4 + 2 * q5)) ...
   * 0.101643953670516041e-19 + Dq1 ^ 2 * sin((q2 + 3 * q3 + 3 * q4 + 2 * q5)) * ...
   0.338813178901720136e-20 + Dq1 ^ 2 * sin((3 * q2 + 3 * q3 + q4 + 2 * q5)) * ...
   0.677626357803440271e-20 + Dq1 ^ 2 * sin((q2 + 3 * q3 + q4 + 2 * q5)) * ...
   (-0.677626357803440271e-20) + Dq1 ^ 2 * sin((-2 * q5 + q4 + q3 + q2)) * ...
   0.149077798716756860e-18 + Dq1 ^ 2 * sin((2 * q5 + q4 + q3 + q2)) * ...
   (-0.216840434497100887e-18) + Dq1 ^ 2 * cos((-q5 + q2)) * ...
   0.216840434497100887e-18 + Dq1 ^ 2 * cos((q5 + q2)) * 0.216840434497100887e-18;
cdq2 = Dq1 ^ 2 * sin((2 * q4 + 2 * q3 + 2 * q2)) * (-0.462956153999997749e-5) + Dq1 ...
   ^ 2 * cos((-q5 + q4 + q3 + 2 * q2)) * 0.284376000000000030e-2 + Dq1 ^ 2 * ...
   cos((q5 + q4 + q3 + 2 * q2)) * (-0.284376000000000030e-2) + Dq1 ^ 2 * cos((-q5 + ...
   q4 + q3)) * 0.216840434497100887e-18 + Dq1 ^ 2 * cos((q5 + q4 + q3)) * ...
   (-0.216840434497100887e-18) + Dq1 ^ 2 * cos((q4 + q3 + 2 * q2)) * ...
   0.777807799999999990e-2 + Dq1 ^ 2 * cos((q4 + q3)) * 0.867361737988403547e-18 + ...
   Dq2 ^ 2 * cos((q4 + q3)) * 0.173472347597680709e-17 + 0.2e-12 * Dq1 * Dq2 * ...
   sin((q4 + q3 + q2)) - 0.2e-12 * Dq1 * Dq3 * sin((q4 + q3 + q2)) + 0.1e-12 * Dq1 ...
   * sin((q4 + q3 + q2)) * Dq4 + Dq1 ^ 2 * sin((2 * q2)) * 0.413642134999999911e0 + ...
   Dq5 ^ 2 * cos((-q5 + q4 + q3)) * (-0.284376000000000030e-2) + Dq5 ^ 2 * cos((q5 ...
   + q4 + q3)) * 0.284376000000000030e-2 + Dq3 ^ 2 * cos((q4 + q3)) * ...
   (-0.777807799999999903e-2) + Dq3 ^ 2 * cos((q5 + q4 + q3)) * ...
   0.284376000000000030e-2 + Dq3 ^ 2 * cos((-q5 + q4 + q3)) * ...
   (-0.284376000000000030e-2) + Dq4 ^ 2 * cos((q4 + q3)) * ...
   (-0.777807799999999903e-2) + Dq4 ^ 2 * cos((q5 + q4 + q3)) * ...
   0.284376000000000030e-2 + Dq4 ^ 2 * cos((-q5 + q4 + q3)) * ...
   (-0.284376000000000030e-2) + Dq1 * Dq5 * cos((q5 + q2)) * ...
   0.568752000000000060e-2 + Dq1 * Dq5 * cos((-q5 + q2)) * ...
   (-0.568752000000000060e-2) + Dq2 * Dq3 * cos((q4 + q3)) * ...
   (-0.155561559999999981e-1) + Dq2 * Dq3 * cos((q5 + q4 + q3)) * ...
   0.568752000000000060e-2 + Dq2 * Dq3 * cos((-q5 + q4 + q3)) * ...
   (-0.568752000000000060e-2) + Dq2 * Dq4 * cos((q4 + q3)) * ...
   (-0.155561559999999981e-1) + Dq2 * Dq4 * cos((q5 + q4 + q3)) * ...
   0.568752000000000060e-2 ...
 + Dq2 * Dq4 * cos((-q5 + q4 + q3)) * (-0.568752000000000060e-2) + Dq2 * Dq5 * ...
   cos((q5 + q4 + q3)) * 0.568752000000000060e-2 + Dq2 * Dq5 * cos((-q5 + q4 + q3)) ...
   * 0.568752000000000060e-2 + Dq3 * Dq4 * cos((q4 + q3)) * ...
   (-0.155561559999999981e-1) + Dq3 * Dq4 * cos((q5 + q4 + q3)) * ...
   0.568752000000000060e-2 + Dq3 * Dq4 * cos((-q5 + q4 + q3)) * ...
   (-0.568752000000000060e-2) + Dq3 * Dq5 * cos((q5 + q4 + q3)) * ...
   0.568752000000000060e-2 + Dq3 * Dq5 * cos((-q5 + q4 + q3)) * ...
   0.568752000000000060e-2 + Dq4 * Dq5 * cos((q5 + q4 + q3)) * ...
   0.568752000000000060e-2 + Dq4 * Dq5 * cos((-q5 + q4 + q3)) * ...
   0.568752000000000060e-2 + Dq1 ^ 2 * sin((2 * q3 + 2 * q2)) * ...
   0.225799867799999984e-1 - 0.1217591948e-3 * sin(q5) * Dq6 * Dq5 + Dq1 * Dq5 * ...
   sin((q4 + q3 + q2)) * (-0.201419266984000003e-2) + Dq1 * Dq5 * sin((2 * q5 + q4 ...
   + q3 + q2)) * 0.528848301320000041e-3 + Dq1 * Dq5 * sin((-2 * q5 + q4 + q3 + ...
   q2)) * 0.528848301320000041e-3 + 0.5000000000e-14 * Dq1 * Dq6 * sin((-2 * q6 + ...
   q4 + q3 + q2 - q5)) - 0.5000000000e-14 * Dq1 * Dq6 * sin((-2 * q6 + q4 + q3 + q2 ...
   + q5)) - 0.1000000000e-13 * Dq2 * Dq6 * sin((2 * q5)) + Dq2 * Dq5 * sin((2 * ...
   q5)) * 0.105769660249000006e-2 + Dq3 * Dq5 * sin((2 * q5)) * ...
   0.105769660274000017e-2 + 0.5000000000e-14 * Dq3 * Dq6 * sin((-2 * q6 + 2 * q5)) ...
   + 0.5000000000e-14 * Dq3 * Dq6 * sin((2 * q6)) + 0.5000000000e-14 * Dq3 * Dq6 * ...
   sin((2 * q5)) + Dq4 * Dq5 * sin((2 * q5)) * 0.105769660279000010e-2 - ...
   0.5000000000e-14 * Dq4 * Dq6 * sin((-2 * q6 + 2 * q5)) - 0.5000000000e-14 * Dq4 ...
   * Dq6 * sin((2 * q6)) + 0.5000000000e-14 * Dq4 * Dq6 * sin((2 * q5)) + 0.2e-12 * ...
   Dq1 * Dq2 * sin((2 * q5 + q4 + q3 + q2)) - 0.2e-12 * Dq1 * Dq3 * sin((-2 * q5 + ...
   q4 + q3 + q2)) - 0.5e-12 * Dq1 * Dq4 * sin((-2 * q5 + q4 + q3 + q2)) - ...
   0.6087959740e-4 * Dq1 * Dq6 * sin((-q5 + q4 + q3 + q2)) + 0.6087959740e-4 * Dq1 ...
```

```
    * Dq6 * sin((q5 + q4 + q3 + q2)) + Dq1 ^ 2 * sin((-2 * q5 + 2 * q4 + 2 * q3 + 2 ...
    * q2)) * (-0.132212075330000010e-3) + Dq1 ^ 2 * sin((2 * q5 + 2 * q4 + 2 * q3 + ...
    2 * q2)) * (-0.132212075330000010e-3);
 cdq3 = Dq1 ^ 2 * sin((2 * q4 + 2 * q3 + 2 * q2)) * (-0.462956153999999104e-5) + Dq1 ...
    ^ 2 * cos((-q5 + q4 + q3 + 2 * q2)) * 0.142188000000000015e-2 + Dq1 ^ 2 * ...
    cos((q5 + q4 + q3 + 2 * q2)) * (-0.142188000000000015e-2) + Dq1 ^ 2 * cos((-q5 + ...
    q4 + q3)) * 0.142188000000000015e-2 + Dq1 ^ 2 * cos((q5 + q4 + q3)) * ...
    (-0.142188000000000015e-2) + Dq2 ^ 2 * cos((-q5 + q4 + q3)) * ...
    0.284376000000000030e-2 + Dq2 ^ 2 * cos((q5 + q4 + q3)) * ...
    (-0.284376000000000030e-2) + Dq1 ^ 2 * cos((q4 + q3 + 2 * q2)) * ...
    0.388903900000000038e-2 + Dq1 ^ 2 * cos((q4 + q3)) * 0.388903900000000038e-2 + ...
    Dq2 ^ 2 * cos((q4 + q3)) * 0.777807800000000077e-2 - 0.3e-12 * Dq1 * sin((q4 + ...
    q3 + q2)) * Dq4 + 0.2257998678e-1 * Dq1 ^ 2 * sin((2 * q3 + 2 * q2)) - ...
    0.1217591948e-3 * sin(q5) * Dq6 * Dq5 + Dq1 * Dq5 * sin((q4 + q3 + q2)) * ...
    (-0.201419266984000046e-2) + Dq1 * Dq5 * sin((2 * q5 + q4 + q3 + q2)) * ...
    0.528848301320000041e-3 + Dq1 * Dq5 * sin((-2 * q5 + q4 + q3 + q2)) * ...
    0.528848301320000149e-3 + 0.5000000000e-14 * Dq1 * Dq6 * sin((-2 * q6 + q4 + q3 ...
    + q2 - q5)) - 0.5000000000e-14 * Dq1 * Dq6 * sin((-2 * q6 + q4 + q3 + q2 + q5)) ...
    - 0.1000000000e-13 * Dq2 * Dq6 * sin((2 * q5)) + Dq2 * Dq5 * sin((2 * q5)) * ...
    0.105769660249000006e-2 + Dq3 * Dq5 * sin((2 * q5)) * 0.105769660249000006e-2 + ...
    0.5000000000e-14 * Dq3 * Dq6 * sin((-2 * q6 + 2 * q5)) + 0.5000000000e-14 * Dq3 ...
    * Dq6 * sin((2 * q6)) + 0.5000000000e-14 * Dq3 * Dq6 * sin((2 * q5)) + Dq4 * Dq5 ...
    * sin((2 * q5)) * 0.105769660274000017e-2 - 0.5000000000e-14 * Dq4 * Dq6 * ...
    sin((-2 * q6 + 2 * q5)) - 0.5000000000e-14 * Dq4 * Dq6 * sin((2 * q6)) + ...
    0.5000000000e-14 * Dq4 * Dq6 * sin((2 * q5)) - 0.5e-12 * Dq1 * Dq2 * sin((-2 * ...
    q5 + q4 + q3 + q2)) + 0.1e-12 * Dq1 * Dq2 * sin((2 * q5 + q4 + q3 + q2)) + ...
    0.1e-12 * Dq1 * Dq4 * sin((2 * q5 + q4 + q3 + q2)) - 0.2e-12 * Dq1 * Dq4 * ...
    sin((-2 * q5 + q4 + q3 + q2)) - 0.6087959740e-4 * Dq1 * Dq6 * sin((-q5 + q4 + q3 ...
    + q2)) + 0.6087959740e-4 * Dq1 * Dq6 * sin((q5 + q4 + q3 + q2)) + Dq1 ^ 2 * ...
    sin((-2 * q5 + 2 * q4 + 2 * q3 + 2 * q2)) * (-0.132212075330000010e-3) + Dq1 ^ 2 ...
    * sin((2 * q5 + 2 * q4 + 2 * q3 + 2 * q2)) * (-0.132212075330000010e-3);
 cdq4 = Dq1 ^ 2 * sin((2 * q4 + 2 * q3 + 2 * q2)) * (-0.462956153999999104e-5) + Dq1 ...
    ^ 2 * cos((-q5 + q4 + q3 + 2 * q2)) * 0.142188000000000015e-2 + Dq1 ^ 2 * ...
    cos((q5 + q4 + q3 + 2 * q2)) * (-0.142188000000000015e-2) + Dq1 ^ 2 * cos((-q5 + ...
    q4 + q3)) * 0.142188000000000015e-2 + Dq1 ^ 2 * cos((q5 + q4 + q3)) * ...
    (-0.142188000000000015e-2) + Dq2 ^ 2 * cos((-q5 + q4 + q3)) * ...
    0.284376000000000030e-2 + Dq2 ^ 2 * cos((q5 + q4 + q3)) * ...
    (-0.284376000000000030e-2) + Dq1 ^ 2 * cos((q4 + q3 + 2 * q2)) * ...
    0.388903900000000038e-2 + Dq1 ^ 2 * cos((q4 + q3)) * 0.388903900000000038e-2 + ...
    Dq2 ^ 2 * cos((q4 + q3)) * 0.777807800000000077e-2 - 0.3e-12 * Dq1 * sin((q4 + ...
    q3 + q2)) * Dq4 - 0.1217591948e-3 * sin(q5) * Dq6 * Dq5 + Dq1 * Dq5 * sin((q4 + ...
    q3 + q2)) * (-0.201419266984000046e-2) + Dq1 * Dq5 * sin((2 * q5 + q4 + q3 + ...
    q2)) * 0.528848301320000041e-3 + Dq1 * Dq5 * sin((-2 * q5 + q4 + q3 + q2)) * ...
    0.528848301320000149e-3 + 0.5000000000e-14 * Dq1 * Dq6 * sin((-2 * q6 + q4 + q3 ...
    + q2 - q5)) - 0.5000000000e-14 * Dq1 * Dq6 * sin((-2 * q6 + q4 + q3 + q2 + q5)) ...
    - 0.1000000000e-13 * Dq2 * Dq6 * sin((2 * q5)) + Dq2 * Dq5 * sin((2 * q5)) * ...
    0.105769660249000006e-2 + Dq3 * Dq5 * sin((2 * q5)) * 0.105769660249000006e-2 + ...
    0.5000000000e-14 * Dq3 * Dq6 * sin((-2 * q6 + 2 * q5)) + 0.5000000000e-14 * Dq3 ...
    * Dq6 * sin((2 * q6)) + 0.5000000000e-14 * Dq3 * Dq6 * sin((2 * q5)) + Dq4 * Dq5 ...
    * sin((2 * q5)) * 0.105769660274000017e-2 - 0.5000000000e-14 * Dq4 * Dq6 * ...
    sin((-2 * q6 + 2 * q5)) - 0.5000000000e-14 * Dq4 * Dq6 * sin((2 * q6)) + ...
    0.5000000000e-14 * Dq4 * Dq6 * sin((2 * q5)) - 0.5e-12 * Dq1 * Dq2 * sin((-2 * ...
    q5 + q4 + q3 + q2)) + 0.1e-12 * Dq1 * Dq2 * sin((2 * q5 + q4 + q3 + q2)) + ...
    0.1e-12 * Dq1 * Dq4 * sin((2 * q5 + q4 + q3 + q2)) - 0.2e-12 * Dq1 * Dq4 * ...
    sin((-2 * q5 + q4 + q3 + q2)) - 0.6087959740e-4 * Dq1 * Dq6 * sin((-q5 + q4 + q3 ...
    + q2)) + 0.6087959740e-4 * Dq1 * Dq6 * sin((q5 + q4 + q3 + q2)) + Dq1 ^ 2 * ...
    sin((-2 * q5 + 2 * q4 + 2 * q3 + 2 * q2)) * (-0.132212075330000010e-3) + Dq1 ^ 2 ...
    * sin((2 * q5 + 2 * q4 + 2 * q3 + 2 * q2)) * (-0.132212075330000010e-3);
 cdq5 = Dq1 ^ 2 * cos((-q5 + q4 + q3 + 2 * q2)) * (-0.142188000000000015e-2) + Dq1 ^ ...
    2 * cos((q5 + q4 + q3 + 2 * q2)) * (-0.142188000000000015e-2) + Dq1 ^ 2 * ...
    cos((-q5 + q4 + q3)) * (-0.142188000000000015e-2) + Dq1 ^ 2 * cos((q5 + q4 + ...
    q3)) * (-0.142188000000000015e-2) + Dq2 ^ 2 * cos((-q5 + q4 + q3)) * ...
    (-0.284376000000000030e-2) + Dq2 ^ 2 * cos((q5 + q4 + q3)) * ...
    (-0.284376000000000030e-2) + Dq1 * Dq2 * sin((q4 + q3 + q2)) * ...
    0.201419266993999990e-2 + Dq1 * Dq3 * sin((q4 + q3 + q2)) * ...
    0.201419266993999990e-2 + Dq1 * sin((q4 + q3 + q2)) * Dq4 * ...
    0.201419266993999990e-2 + sin(q5) * Dq1 ^ 2 * 0.145868160000000007e-2 + Dq1 * ...
    Dq2 * cos((-q5 + q2)) * 0.568752000000000060e-2 + Dq1 * Dq2 * cos((q5 + q2)) * ...
    (-0.568752000000000060e-2) + Dq3 ^ 2 * sin((2 * q5)) * ...
    (-0.528848301320000041e-3) + Dq4 ^ 2 * sin((2 * q5)) * ...
    (-0.528848301320000041e-3) + Dq1 * Dq2 * sin((-2 * q5 + q4 + q3 + q2)) * ...
    (-0.528848301319999933e-3) + Dq1 * Dq2 * sin((2 * q5 + q4 + q3 + q2)) * ...
    (-0.528848301320000041e-3) + Dq1 * Dq3 * sin((-2 * q5 + q4 + q3 + q2)) * ...
    (-0.528848301320000041e-3) + Dq1 * Dq3 * sin((2 * q5 + q4 + q3 + q2)) * ...
```

```
   (−0.528848301320000041e−3) + Dq1 * Dq4 * sin((2 * q5 + q4 + q3 + q2)) * ...
   (−0.528848301320000041e−3) + Dq1 * Dq4 * sin((−2 * q5 + q4 + q3 + q2)) * ...
   (−0.528848301319999933e−3) + 0.6087959740e−4 * Dq1 * Dq6 * sin((−q5 + q4 + q3 + ...
   q2)) + 0.6087959740e−4 * Dq1 * Dq6 * sin((q5 + q4 + q3 + q2)) + 0.1e−13 * Dq1 * ...
   Dq6 * sin((q4 + q3 + q2)) − 0.1e−13 * Dq1 * Dq6 * sin((−2 * q6 + q4 + q3 + q2)) ...
   + Dq2 * Dq3 * sin((2 * q5)) * (−0.105769660264000008e−2) + Dq2 * Dq4 * sin((2 * ...
   q5)) * (−0.105769660264000008e−2) + 0.1217591948e−3 * Dq2 * Dq6 * sin(q5) + Dq3 ...
   * Dq4 * sin((2 * q5)) * (−0.105769660264000008e−2) + 0.1217591948e−3 * Dq3 * Dq6 ...
   * sin(q5) + 0.1e−13 * Dq3 * Dq6 * sin((−2 * q6 + q5)) + 0.1217591948e−3 * Dq6 * ...
   Dq4 * sin(q5) − 0.1e−13 * Dq6 * Dq4 * sin((−2 * q6 + q5)) + Dq1 ^ 2 * sin((−2 * ...
   q5 + 2 * q4 + 2 * q3 + 2 * q2)) * 0.132212075330000010e−3 + Dq1 ^ 2 * sin((2 * ...
   q5)) * 0.264424150660000021e−3 + Dq1 ^ 2 * sin((2 * q5 + 2 * q4 + 2 * q3 + 2 * ...
   q2)) * (−0.132212075330000010e−3) + Dq2 ^ 2 * sin((2 * q5)) * ...
   (−0.528848301319999933e−3);
cdq6 = −0.6087959740e−4 * Dq1 * Dq2 * sin(q5 + q4 + q3 + q2) + 0.6087959740e−4 * ...
   Dq1 * Dq2 * sin(−q5 + q4 + q3 + q2) − 0.6087959740e−4 * Dq1 * Dq3 * sin(q5 + q4 ...
   + q3 + q2) + 0.6087959740e−4 * Dq1 * Dq3 * sin(−q5 + q4 + q3 + q2) − ...
   0.6087959740e−4 * Dq1 * Dq4 * sin(q5 + q4 + q3 + q2) + 0.6087959740e−4 * Dq1 * ...
   Dq4 * sin(−q5 + q4 + q3 + q2) − 0.6087959740e−4 * Dq5 * Dq1 * sin(−q5 + q4 + q3 ...
   + q2) − 0.6087959740e−4 * Dq5 * Dq1 * sin(q5 + q4 + q3 + q2) − 0.1217591948e−3 * ...
   Dq5 * sin(q5) * Dq4 − 0.1217591948e−3 * Dq5 * sin(q5) * Dq2 − 0.1217591948e−3 * ...
   Dq5 * sin(q5) * Dq3;

%Setting up the matrices
M = [m11 m12 m13 m14 m15 m16;m21 m22 m23 m24 m25 m26;m31 m32 m33 m34 m35 m36;m41 ...
   m42 m43 m44 m45 m46;m51 m52 m53 m54 m55 m56;m61 m62 m63 m64 m65 m66];
G = [g1;g2;g3;g4;g5;g6];
CDq = [cdq1;cdq2;cdq3;cdq4;cdq5;cdq6];
```

Appendix C

Simulation Experiment Implementation

The simulation experiment contains too much code to be reproduced here to its full extent, but a selection of the most central functions is given in this appendix. The author can be contacted for the rest of the code.

C.1. Simulink Diagrams

The top-level block diagram is shown in Figure 7.1. The first subsystem, the trajectory generator, is shown in Figure C.1. The second subsystem, which contains the inverse dynamics controller and force estimator, is shown in Figure C.2. The last subsystem, which generates collision forces and implement the forward dynamics of the UR5 manipulator, is shown in Figure C.3.

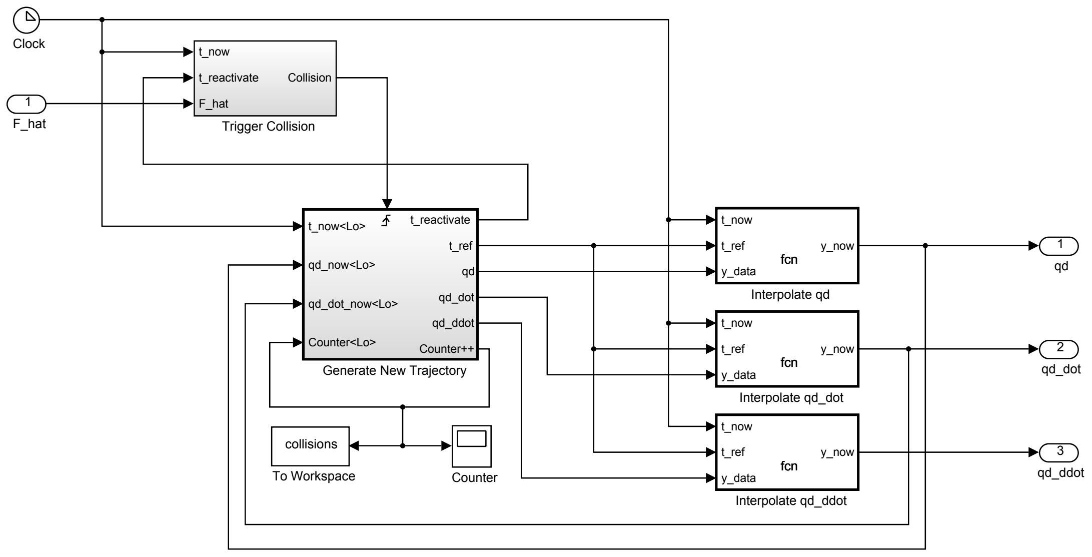


Figure C.1: Trajectory generator

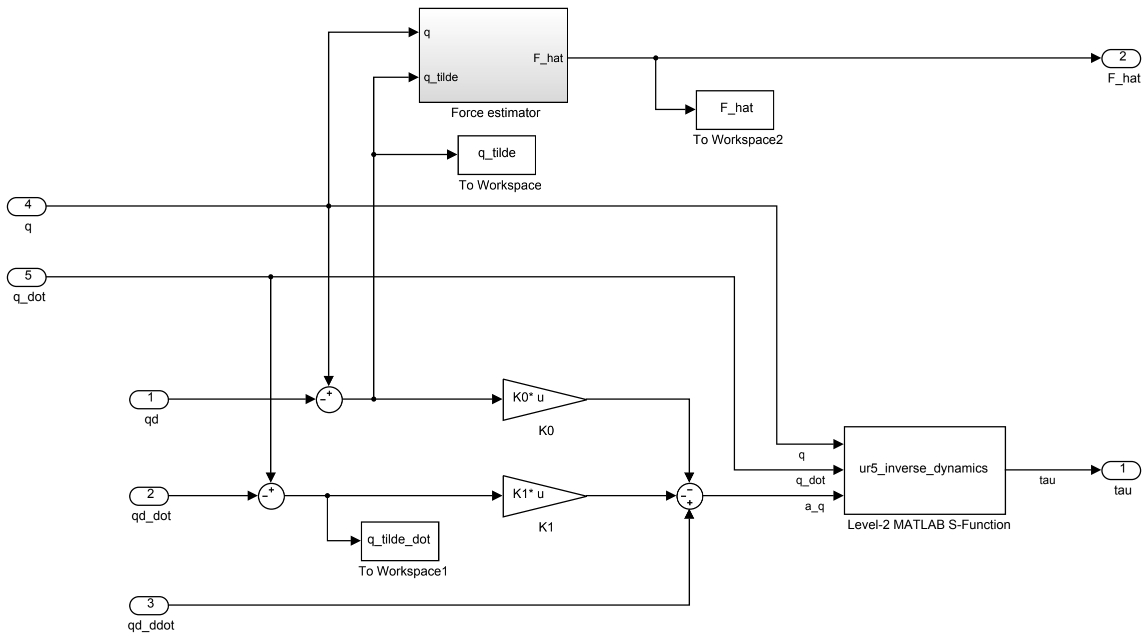


Figure C.2: Inverse dynamics controller and force estimator

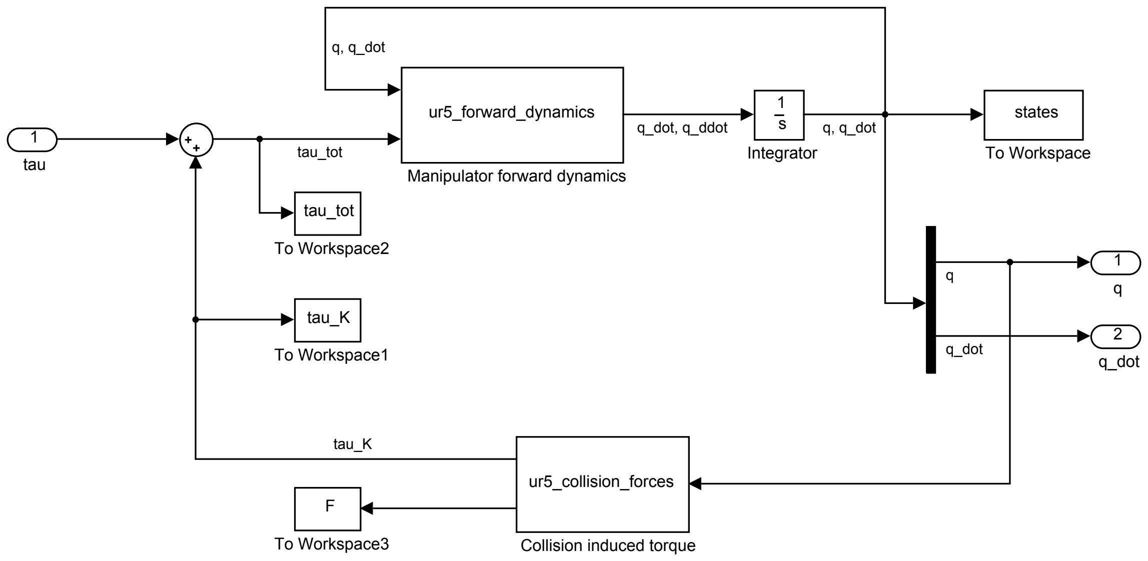


Figure C.3: UR5 model dynamics

C.2. Matlab Code

Some of the code requires the robotics toolbox (Corke, 2011) to be installed. The functions are called from the Simulink model either as S-functions or as interpreted Matlab functions.

C.2.1. UR5 Forward Dynamics (S-function)

The usage of this function can be seen in Figure C.3. The inverse dynamics controller and force estimator is implemented in a similar way.

```matlab
function ur5_forward_dynamics(block)
%SFUNDSC2_LEVEL2 Level-2 version of sfundsc2.m
%
%   Copyright 2003-2007 The MathWorks, Inc.

%%
%% The setup method is used to setup the basic attributes of the
%% S-function such as ports, parameters, etc. Do not add any other
%% calls to the main body of the function.
%%
setup(block);

%endfunction

function setup(block)

  % Register number of ports
  block.NumInputPorts  = 2;
  block.NumOutputPorts = 1;

  % Setup port properties to be inherited or dynamic
  block.SetPreCompInpPortInfoToDynamic;
  block.SetPreCompOutPortInfoToDynamic;

  % Override input port properties
  block.InputPort(1).Dimensions  = [12 1];
  block.InputPort(2).Dimensions  = [6 1];
  block.InputPort(1).DirectFeedthrough = false;
  block.InputPort(2).DirectFeedthrough = false;

  % Override output port properties
  block.OutputPort(1).Dimensions  = [12 1];

  % Register parameters
  block.NumDialogPrms = 0;

  % Register sample times
  block.SampleTimes = [-1 0];


  %% -----------------------------------------------------------------
  %% Options
  %% -----------------------------------------------------------------
  % Specify if Accelerator should use TLC or call back into
  % M-file
  block.SetAccelRunOnTLC(false);


  %% -----------------------------------------------------------------
  %% Register methods
  %% -----------------------------------------------------------------
  block.RegBlockMethod('InitializeConditions', @InitializeConditions);
  block.RegBlockMethod('Outputs', @Outputs);

%endfunction
```

```
%% ---------------------------------------------------------------------------
%% The local functions
%% ---------------------------------------------------------------------------

function InitializeConditions(block)
block.InputPort(1).Data(1) = 0;
block.InputPort(1).Data(2) = 0;
block.InputPort(1).Data(3) = 0;
block.InputPort(1).Data(4) = 0;
block.InputPort(1).Data(5) = 0;
block.InputPort(1).Data(6) = 0;
block.InputPort(1).Data(7) = 0;
block.InputPort(1).Data(8) = 0;
block.InputPort(1).Data(9) = 0;
block.InputPort(1).Data(10) = 0;
block.InputPort(1).Data(11) = 0;
block.InputPort(1).Data(12) = 0;

function Outputs(block)

%Defining input ports
u = block.InputPort(2).Data(1:6);
q = block.InputPort(1).Data(1:6);
Dq = block.InputPort(1).Data(7:12);

[ M, G, CDq ] = ur5_dynamics( q, Dq );

%Computing the model
DDq = M\(-CDq-G+u);


%Sending to output port
block.OutputPort(1).Data = [Dq; DDq];
```

C.2.2. Trajectory Generator

This function is run after every collision detected and generates a new trajectory for the controller.

```
function y = ur5_generate_trajectory( u )
%Returns a new desired trajecory after a collision is detected.
% u = [t, q, q_dot]
% y = [t_ref, qd, qd_dot, qd_ddot]

global I ur5 PathGen

t0 = u(1);
q0 = u(2:7);
q0_dot = u(8:13);
initializing = 0;

if q0==zeros(6,1)
   q0 = [ 2.5511   -1.1193    2.5706    0.1196    1.5708   -0.5905 ]';
   initializing = 1;
end

R = [0 1 0; 1 0 0; 0 0 -1];

%% Find current position in collision-space

[J, T] = ur5_jacobian( q0 );

[~,pos_w] = tr2rt(T);
vel_w = J(1:3,:)*q0_dot;

% From workspace xy-coordinates to collision-space
pos_c  = zeros(1,2);
pos_c(1) = round ( 1 / 0.0263 * ( pos_w(1) - 0.1238 ) );
pos_c(2) = round ( 1 / 0.0263 * ( pos_w(2) + 0.2762 ) );

% Add current point to collision map unless we are initializing
if ~initializing

    if I(pos_c(1),pos_c(2)) == −1
        disp('Warning: Collision already happened here.');
    end

    I(pos_c(1),pos_c(2)) = −1;

    if abs( vel_w(1) ) > abs( vel_w(2) )
        % collision in x−direction
        pos_c(1) = pos_c(1) − sign( vel_w(1) );
    else
        % collision in y−direction
        pos_c(2) = pos_c(2) − sign( vel_w(2) );
    end

end

pos_c(1) = max(pos_c(1),2);


%% Run the Hopfield neural network to generate a new path

disp(['Collision at time: ' num2str(t0) ':']);
disp('q(t):');
disp(q0);

[trajectory, ~,path] = hopfield_path_planner( pos_c(1), pos_c(2), 18, 7, I );
PathGen(:,:,end+1)=path;


%% From collision space coordinates of neural network to workspace xy−coordinates
% x_w = 0.0263*x_g + 0.1238;
% y_w = 0.0263*y_g − 0.2762;

rows = size(trajectory,1);
trajectory(:,1) = 0.0263 * (trajectory(:,1)) + 0.1238;
trajectory(:,2) = 0.0263 * (trajectory(:,2)) − 0.2762;
trajectory(:,3) = zeros(rows,1);

R0(:,:,1:rows+1) = reshape(repmat(R,1,rows+1),3,3,rows+1);

T = rt2tr(R0, [ pos_w'; trajectory]');

% Waypoints to configuration space
Q = ur5.ikine(T,q0,'tol',1e−4,'ilimit',15000);


%% Generate full trajectory by connecting smooth splines
num_rows = 5000;
m = 100;
T = round(size(Q,1)/2);
dt = 1/((size(Q,1)−1)*m)*T;
t = (0:dt:num_rows*dt−1e−10) + t0;

[qt,qdt,qddt] = jtrajcombine( Q, m, num_rows, q0_dot' );

disp('New trajectory generated.')

y = [t', qt, qdt, qddt];


end
```

C.2.3. Hopfield Neural Network Path Planner

This function implements the Hopfield neural network of chapter 6. In the first stage it iterates the neural network until convergence, and in the second stage it finds the shortest path by following the steepest ascent along connected neurons.

```
function [trajectory, sigma, path] = hopfield_path_planner( x0, y0, x_d, y_d, I )
%Returns the shortest path from (x0,y0) to (x_d,y_d) in the neuronal space
%defined by I. I is an n*m matrix with value of -1 at obstructions and +1
%value at destination, free space must have value 0.
%   Trajectory returns a set of (u,v) coordinates from initial point to
%destination. Sigma returns the final state values of the neural network.
%Path returns an n*m matrix which can be plotted to see the path in
%neuronal space.

[n,m] = size(I);

T = sqrt(n*m) * 10;

sigma = zeros( n, m );
sigma_next = zeros( n, m );

i_dest = x_d;
j_dest = y_d;

I(i_dest,j_dest)=1;
I(:,1) = -1;
I(:,m) = -1;
I(1,:) = -1;
I(n,:) = -1;
I = I*1000;

i_start = x0;
j_start = y0;

% Iterate neural network
for t = 1:T-1

    % For every neuron
    for i=2:n-1
        for j=2:m-1

            % Update neuron
            x = I(i,j);
            for index=[-1 0; 1 0; 0 -1; 0 1]'
                x = x + sigma(i+index(1),j+index(2));
            end
            sigma_next(i,j) = sigmoid( x );

        end
    end

    sigma = sigma_next;
end


finished = 0;
t = 1;
i = i_start;
j = j_start;

path = I / 1000 + 1;
path(i,j) = .75;

trajectory(1,1:2) = [i, j];


% Find optimal path along gradient ascent
while finished == 0
    move = [0,0];
    v = -1;
    for index=[-1 0; 1 0; 0 -1; 0 1]'
        if sigma(i+index(1),j+index(2)) >= v
            v = sigma(i+index(1),j+index(2));
            move = index';
        end
    end


    i = i + move(1);
    j = j + move(2);
    t = t + 1;
    trajectory(t,:) = [i, j];
    path(i,j) = .5;


    if ((move(1) == 0 && move(2) == 0) || i == 1 || i==n || j==1 || j==m) || ...
       size(trajectory,1)>1200
        finished = 1;
        display('Error: Does not converge');
    end

    if i==i_dest && j == j_dest
        finished = 1;
        display('Hooray!');
    end

end

sigma(:,:,2) = path(:,:);


function y = sigmoid(x)
    beta = 0.1;
    y = clamp(beta*x,0,1);
end
function y = clamp(x,a,b)
    y = min(max(x,a),b);
end
end
```

Bibliography

Peter I. Corke. *Robotics, Vision & Control: Fundamental Algorithms in Matlab*. Springer, 2011. ISBN 978-3-642-20143-1. URL `http://www.petercorke.com/Robotics_Toolbox.html`.

A. De Luca, A. Albu-Schaffer, S. Haddadin, and G. Hirzinger. Collision detection and safe reaction with the dlr-iii lightweight manipulator arm. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 1623–1630. IEEE, 2006.

O. Egeland and J.T. Gravdahl. *Modeling and simulation for automatic control*, volume 76. Marine Cybernetics, 2002.

Roy Featherstone and David E. Orin. Dynamics. In Bruno Siciliano and Oussama Khatib, editors, *Springer Handbook of Robotics*, pages 35–65. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-30301-5. URL `http://dx.doi.org/10.1007/978-3-540-30301-5_3`.

R. Glasius, A. Komoda, and S.C.A.M. Gielen. Neural network dynamics for path planning and obstacle avoidance. *Neural Networks*, 8(1):125–133, 1995.

PJ Hacksel and SE Salcudean. Estimation of environment forces and rigid-body velocities using observers. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 931–936. IEEE, 1994.

S. Haddadin, A. Albu-Schaffer, A. De Luca, and G. Hirzinger. Collision detection and reaction: A contribution to safe physical human-robot interaction. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 3356 –3363, sept. 2008. doi: 10.1109/IROS.2008.4650764.

Simon Haykin. *Neural Networks: A Comprehensive Foundation (2nd Edition)*. Prentice Hall, 2 edition, July 1998. ISBN 0132733501.

Herman Høifødt. Dynamic modeling and simulation of robot manipulators : The newton-euler formulation, 2011.

John M. Hollerbach. A recursive lagrangian formulation of maniputator dynamics and a comparative study of dynamics formulation complexity. *Systems, Man and Cybernetics, IEEE Transactions on*, 10(11):730 –736, nov. 1980. ISSN 0018-9472. doi: 10.1109/TSMC.1980.4308393.

R. Holmberg and O. Khatib. Development and control of a holonomic mobile robot for mobile manipulation tasks. *The International Journal of Robotics Research*, 19(11): 1066–1074, 2000.

Dushyant Palejiya and Herbert G. Tanner. Hybrid velocity/force control for robot navigation in compliant unknown environments. *Robotica*, 24:745–758, 2006. ISSN 1469-8668. doi: 10.1017/S026357470600292X. URL `http://dx.doi.org/10.1017/S026357470600292X`.

William M. Silver. On the equivalence of lagrangian and newton-euler dynamics for manipulators. *The International Journal of Robotics Research*, 1(2):60–70, 1982. doi: 10.1177/027836498200100204. URL `http://ijr.sagepub.com/content/1/2/60.abstract`.

M.W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot modeling and control.* John Wiley & Sons, Hoboken, NJ, 2006.

A. Stolt, M. Linderoth, A. Robertsson, and R. Johansson. Force controlled robotic assembly without a force sensor. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 1538–1543. IEEE, 2012.

K. Suita, Y. Yamada, N. Tsuchida, K. Imai, H. Ikeda, and N. Sugimoto. A failure-to-safety ”kyozon” system with simple contact detection and stop capabilities for safe human-autonomous robot coexistence. In *Robotics and Automation, 1995. Proceedings., 1995 IEEE International Conference on*, volume 3, pages 3089 –3096 vol.3, may 1995. doi: 10.1109/ROBOT.1995.525724.

S. Takakura, T. Murakami, and K. Ohnishi. An approach to collision detection and recovery motion in industrial robot. In *Industrial Electronics Society, 1989. IECON’89., 15th Annual Conference of IEEE*, pages 421–426. IEEE, 1989.

Manual - UR5 with CB2. Universal Robots. Version 1.6.