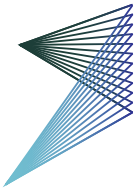
University of Zurich
Department of Informatics



ROBOTICS &
PERCEPTION
GROUP

Benjamin Keiser

Torque Control of a KUKA youBot Arm

Master Thesis

Robotics and Perception Group
University of Zurich

Supervision

UZH:
Elias Müggler
Matthias Fässler
Prof. Dr. Davide Scaramuzza

ETH:
Stephan Huck
Prof. Dr. John Lygeros

September 2013

Contents

Abstract

Accurate end-effector trajectory tracking is a fundamental requirement for many robotic manipulator tasks—even for simple movements such as picking objects straight up from the floor. While end-effector trajectories are given in Cartesian space, we can only control the individual manipulator joints. However, small errors in the joint positions can sum up to large position errors of the end effector in Cartesian space. Furthermore, the change of the end-effector position for a given change of the manipulator joint positions depends on the manipulator configuration. In this project, we aim on using a KUKA youBot for accurate and fast grasping. Existing control schemes for the KUKA youBot arm, namely joint position and joint velocity control, do not consider the arm configuration and its dynamics. Therefore, we designed a torque controller, based on a dynamical model of the KUKA youBot arm, and validated it on a real platform. The designed torque controller allows ten times faster grasping of objects than existing control schemes with similar tracking performance of the grasp trajectory. As an application, we demonstrate autonomous grasping of objects, which are detected using an on-board Kinect-like sensor. With the proposed control scheme, objects can be grasped even when the youBot is moving.

Nomenclature

Notation

g	Gravitational acceleration
$\mathbf{J}$	Jacobian
$\mathbf{M}$	Manipulator Inertia Matrix
$\mathbf{C}$	Manipulator Coriolis Matrix
$\mathbf{n}$	Vector of external forces acting on the manipulator
$\boldsymbol{\tau}$	Joint Torques
$\boldsymbol{\theta}$	Joint Positions
$\mathbf{e}$	Tracking error in joint positions
$\mathbf{K}_v, \mathbf{K}_p$	Controller gain matrices

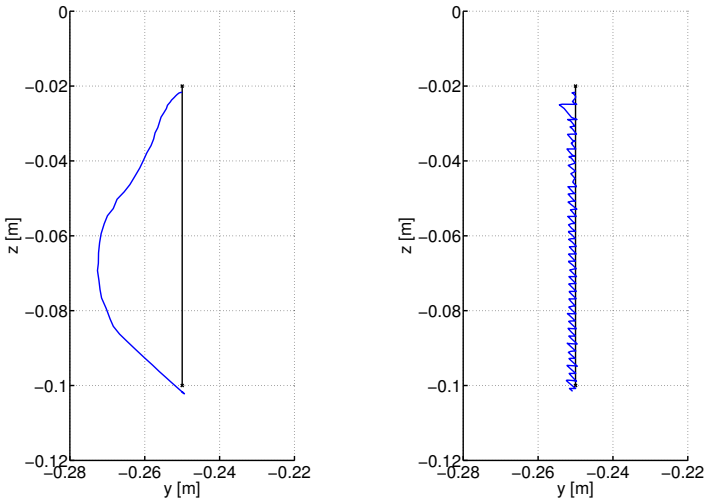
Acronyms and Abbreviations

CTC	Computed-torque Control
DoF	Degrees of Freedom
Orocos	Open Robot Control Software
PCA	Principal Component Analysis
PCL	Point Cloud Library
PID	Proportional-Integral-Derivative
RANSAC	Random Sample Consensus
RGBD	Red Green Blue Depth
ROS	Robot Operating System
RPG	Robotics and Perception Group
SAC	Sample Consensus

Chapter 1

Introduction

A robot is a machine that can interact with its environment. One form of interaction is the manipulation of objects, e.g. picking up objects, moving them around, and putting them down again. Every attempt at object manipulation depends on an accurate and precise manipulator. These two properties enable the end effector to move exactly where the object is.
Conventional robot arms consist of rotatory and prismatic joints. While these can be controlled accurately and precisely in the joint space, it is more difficult to control a robotic arm directly in the Cartesian space. The calculation of the current end-effector pose in the Cartesian space from the joint positions is done using the forward kinematic chain. The inverse problem is a lot harder to compute. Depending on the degrees of freedom of a robot arm, the same end-effector pose might be reached by different joint positions. On the other hand, not every end-effector pose in Cartesian space has a feasible solution in the joint space. Even if all joint positions for a Cartesian space trajectory are known and feasible, there is no guarantee that tracking of the trajectory is precise. Control in joint space does not guarantee smooth motion in Cartesian space. Precise and accurate following of trajectories is important for various reasons. Unpredicted movements may damage the environment or the robot itself, or move objects out of position that have to be picked up. Inaccurate tracking of trajectories will cause the end effector to be out of position. This could cause the robot to be damaged or simply cause a task to become impossible to complete. Especially tasks in dynamic surroundings consisting of moving objects would suffer from inaccurate or imprecise tracking of trajectories.
We now assume that for a feasible Cartesian trajectory, the joint positions for every point on the trajectory are known. As we are considering discrete time control, the number of points will be limited to the number of time-steps. Depending on how we choose to move from one point to the next, tracking of the trajectory will be accurate and smooth or inaccurate with unpredicted manipulator movements. The easiest way to move from one point to the next, is to simply command the joint to change its current position to the next. We call this the position control mode. Another method is to command a certain velocity to a joint until the desired position is reached. This is the so called velocity control mode. For both of these methods, we do not know how the end-effector is moving between two points. A third method is to implement a torque controller, which calculates the required joint torques to switch from

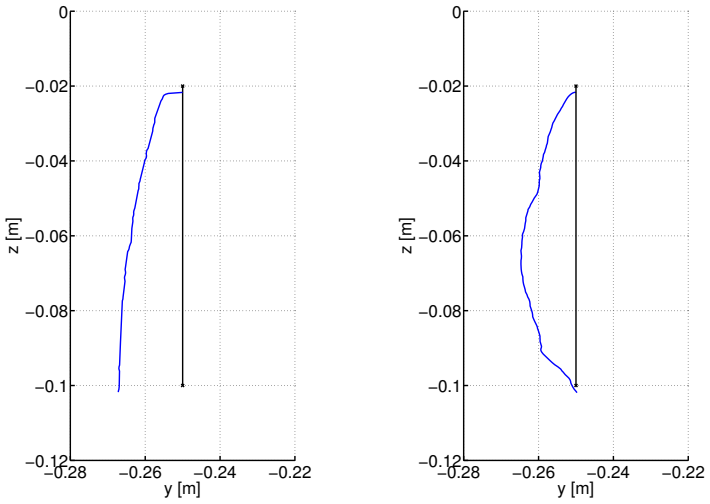


(a) Direct step $\Delta t = 1.5\,\mathrm{s}$ (b) 1 mm step $\Delta t = 15.4\,\mathrm{s}$

Figure 1.1: Left: Lowering of the end effector with direct position command. Right: Lowering of the end effector in 1 mm steps. Both trajectories are compared to the ideal trajectory in form of a straight line.

the current position to the next. The main advantage of the torque controller compared to the other methods is that it also takes the joint configuration and manipulator dynamics into account. This allows for smoother tracking of trajectories.

For the test platform, a KUKA youBot, the position and velocity control methods deliver results which are unsatisfactory for certain tasks such as grasping of objects. As an example object, boxes are chosen. We assume that the best way to grasp a box is to position the end effector directly above the object and then lowering it straight down. In Figure 1.1, it is shown that when lowering the end effector by directly commanding the new joint positions, it does not move in a straight line at all. Such a trajectory is very inconvenient when trying to grasp an object, as it is almost certain that the object will be moved and grasping fails. When commanding the lowering in 1 mm steps, the trajectory is much closer to the straight line. However, the task is taking up to 10 times longer to complete than any other tested method. In addition to the long time required, the end effector is also oscillating by a small measure which is inconvenient. Figure 1.2 shows the trajectory if the lowering is commanded in the velocity control mode. Without a joint position feedback to correct the velocity, the reached position differs vastly from the desired position. Even with a position feedback the manipulator does not move in a straight line. In short, none of the existing control schemes is able to accurately and quickly follow the desired trajectory. That is why we implemented a torque controller based on a dynamical model of the KUKA youBot arm. The torque controller is able to track a Cartesian end-effector trajectory with a low position error while remaining fast. It is successfully used for autonomous grasping of objects and grasp while



(a) Velocity $\Delta t = 1.7\,\mathrm{s}$ (b) Velocity with feedback $\Delta t = 2.92\,\mathrm{s}$

Figure 1.2: Left: Lowering of the end effector in velocity mode without feedback. Right: Lowering of the end effector in velocity mode with feedback. Both trajectories are compared to the ideal trajectory in form of a straight line.

driving.
Grasping objects requires detection of the objects in simple flat scenes. For this the youBot is equipped with a Kinect camera system. A Kinect system is equipped with a color as well as an infra-red camera which allows capturing of an image with a corresponding depth map. These two images can then be combined to form a color point cloud. We developed an algorithm to detect and extract objects and their position and orientation in the point cloud. Afterwards, the robot moves independently to that position and picks up the objects using the designed torque controller.

1.1 Related work

Shashank Sharma, Gerhard K. Kraetzschmar, Christian Scheurer, and Rainer Bischoff developed an unified closed form solution for the inverse kinematics of the youBot which allows fast computation of joint positions for Cartesian end-effector positions [8]. Using this approach, generated Cartesian trajectories are easily transformed to the corresponding joint space trajectories required by the torque controller, as long as all the positions are feasible.
Timothy Jenkel, Richard Kelly and Paul Shepanski show an approach to object manipulation on the KUKA youBot [3]. Their system is able to detect objects autonomously in the environment then pick them up and put them down using a simple user interface. In order to effectively control the youBot arm they make use of the existing ROS arm navigation stack and adapt it for the KUKA youBot. The stack allows to control the manipulator in Cartesian space based

on the implemented inverse kinematic chain. After objects get detected using a Microsoft Kinect attached to the arm, the youBot is then navigated to a position from where the objects are grasped using a set of overhead cameras. For grasping, they make use of the ROS object manipulation stack, which provides a framework for pick and place actions. Contrary to this thesis, they make use of external stacks for efficient manipulator control, grasping of objects and even use an external camera system to track the base. In this project, we are developing a torque controller which is designed for close tracking of trajectories and assume grasping is always optimal when attempted from directly above. In addition, the base is only controlled using the odometry messages generated by the KUKA youBot ROS stack.
The object detection algorithm was developed by Paul Malmsten [5] and is very similar to the one implemented in this thesis (see Chapter 5). The main difference is that their algorithm is using iterative closest point to match objects to a database, while in this thesis no database is used and objects are simply detected by finding an appropriate bounding box.

Chapter 2

Hardware

We want to use a KUKA youBot to autonomously grasp objects using a torque controller and an object detection algorithm. The youBot consists of a mobile omnidirectional base and a 5 DoF arm, which are described in Section 2.1. To generate a point cloud for the object detection algorithm, we use an ASUS Xtion PRO sensor, which is described in Section 2.2.

2.1 KUKA youBot

The robot used in this thesis is a KUKA youBot [1]. It consists of a 5 DoF arm with a two-finger gripper, mounted on an omnidirectional platform. It was developed for use in research and education. Communication of the base and the arm of the youBot is enabled via EtherCAT which has real time capabilities.

Figure 2.1: The base of the KUKA youBot with omnidirectional wheels.

Figure 2.2: The 5 DoF arm of the KUKA youBot

2.1.1 youBot Arm

The arm of the KUKA youBot (see Figure 2.2) consists of five rotatory joints and a two-finger gripper as an end effector. It reaches a height of 655 mm, has a weight of 6.3 kg and is designed to carry a payload of up to 0.5 kg in weight. The rotatory joints of the arm rotate around two different axes. Joints two, three, and four are rotating in parallel around one axis and joints one and five in parallel around the other if the arm is pointing straight up. The maximum rotation speed of a joint is 90 deg /s and the end effector can be opened 11.5 mm per finger.

2.1.2 youBot Base

The base of the KUKA youBot is pictured in Figure 2.1. It consists of a mini PC running Ubuntu, a battery and four omnidirectional wheels with separate motors. These allow the youBot to turn on the spot as well as move in any direction without having to turn first. The maximum speed of the base is $0.8\,\frac{\mathrm{m}}{\mathrm{s}}$. It has a weight of 20 kg, and measures 580 mm by 380 mm by 140 mm.

2.2 The ASUS Xtion PRO

The ASUS Xtion PRO[1] works the same way as the better known Microsoft Kinect. Image 2.3 shows the sensor mounted on the KUKA youBot. The sensor was developed by PrimeSense[2] and both ASUS and Microsoft bought a license for the usage of that sensor. In addition to an RGB camera with up to SXGA

[1] http://www.asus.com/Multimedia/Xtion_PRO_LIVE/
[2] http://www.primesense.com/

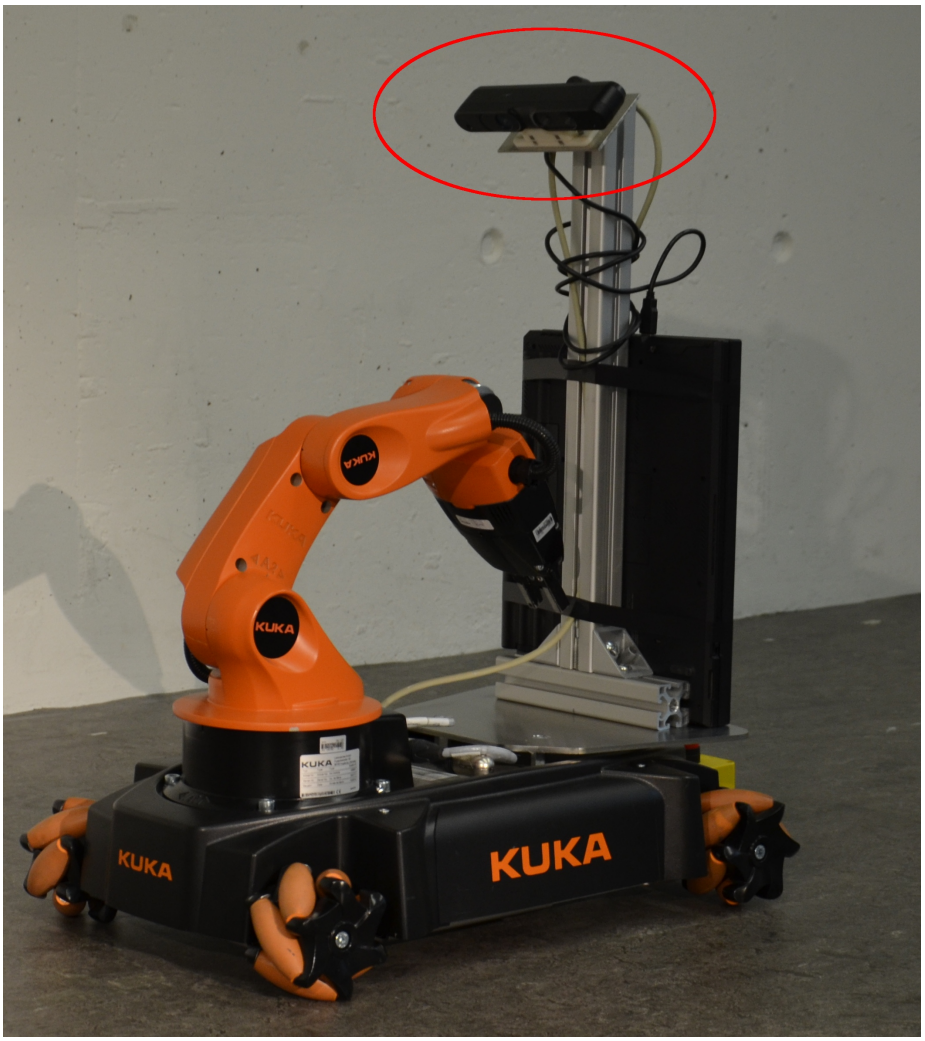


Figure 2.3: ASUS Xtion PRO mounted on the KUKA youBot

resolution (1280x1024 pixels), it also has a depth sensor made for distances between 0.8 m and 3.5 m. The depth measurement is done by combining a laser with an infra-red sensitive camera. The laser is directing its beam at points in a raster, the infra-red camera registers these points and calculates the distance from the camera for each point. This results in the output of a VGA (640x480 pixels) depth map. The depth map and RGB image are then combined to output an RGBD point cloud. Such a point cloud consists of points for which the x, y, and z position as well as its color is known.

Chapter 3

Software

All the software for this project was developed on Ubuntu using the Robot Operating System (ROS). After initially using Orocos, we switched to ROS when issues with over currents on the youBot joint controllers arose and official support was dropped (see Appendix A). This chapter briefly introduces ROS, a simulation environment we use for the youBot as well as additional libraries used.

3.1 ROS

ROS[1] is an environment that facilitates the development of robotics applications. It includes libraries and tools to provide hardware abstraction, device drivers, visualizers, message-passing and so on. An advantage of ROS is that it is very transparent. Programs are built as ROS nodes, which connect to a single ROS master. Every node connected to this master can then listen to all the messages provided by other nodes by simply subscribing to the corresponding topics. In addition to messages, parameters and services can be made available for all nodes connected to the master in the same way. The user can interact with the master by a series of simple commands. This way, the user can publish messages or call services manually. There is an existing ROS component for the KUKA youBot, which builds an interface for communication between ROS and the youBot driver which accesses the hardware directly. This provided youBot component handles messages to command joint positions or velocities. It also publishes information about the current state the youBot is in. For this thesis, the component was extended to additionally allow the handling of torque messages. ROS is still under active development and new versions are released regularly. The version used in this thesis is ROS fuerte.

3.2 Webots

Webots[2] is a development environment for robotics which is used to create advanced simulations. It comes with an implemented model of the KUKA

[1] `http://wiki.ros.org/`
[2] `http://www.cyberbotics.com/overview`

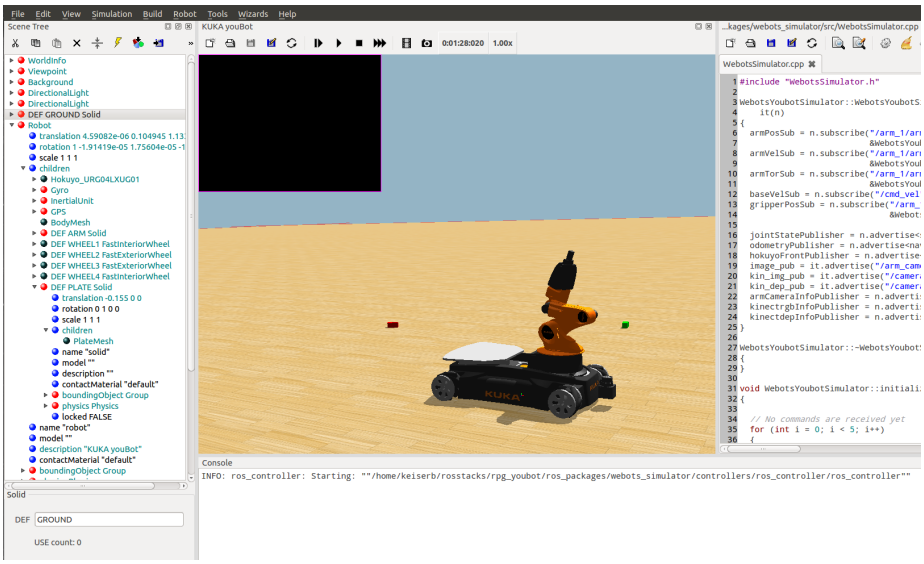


Figure 3.1: Example of a Webots simulation with the KUKA youBot and graspable objects in the background.

youBot. Webots supports physics supplied by plug-ins, which are adapted for different cases. Figure 3.1 shows the main window of a Webots simulation. On the left is the scene tree which consists of all the elements in the scene. The console output can be seen at the bottom while on the right side, there is a text editor where code can be adapted directly. We integrated Webots in our ROS code so it sends and receives the same messages as the physical robot.

3.3 PCL

The Point Cloud Library (PCL) is a standalone C++ library to facilitate the handling of point clouds and was originally developed as a ROS component [7]. Its applications include filtering, feature estimation, surface reconstruction, registration, and segmentation. E.g. a voxel grid filter is used to downsample point clouds, while a segmentation algorithm based on RANSAC is able to find planes in the point cloud. Basically the PCL is to point clouds what OpenCV is to images. In this thesis we use the PCL to develop an algorithm to detect graspable objects and compute their pose.

3.4 Fuzzylite

Fuzzylite[3] is a library that provides the user with an environment to easily create fuzzy logic controllers. It includes a graphical user interface which is used to create the controller logics, which can then be simply exported into C++ code. For an example of how this looks, see Chapter 4.5. In this project it is used to test self tuning of the developed torque controller.

[3] `http://code.google.com/p/fuzzylite/`

Chapter 4

Torque control

The movement, which an arm joint will perform when a certain torque is applied, depends on the robot dynamics. In order to have our manipulator follow a specific trajectory closely, a complete and accurate model of the robot dynamics is necessary. With this model, the required joint torques for given joint positions, velocities, and accelerations can be calculated. In this chapter we present the dynamical model of the youBot as well as the control laws used to follow trajectories. Both Section 4.1 and Section 4.2 follow the proceedings from the book "A Mathematical Introduction to Robotic Manipulation"[6].

4.1 Dynamical model

The dynamical model of a robotic manipulator can be written as

$$\boldsymbol{\tau} = \mathbf{M}(\boldsymbol{\theta})\ddot{\boldsymbol{\theta}} + \mathbf{C}(\boldsymbol{\theta}, \dot{\boldsymbol{\theta}})\dot{\boldsymbol{\theta}} + \mathbf{n}(\boldsymbol{\theta}). \tag{4.1}$$

For a robotic arm with n joints, it is defined by the manipulator inertia matrix $\mathbf{M} \in \mathbb{R}^{n\times n}$, the Coriolis matrix $\mathbf{C} \in \mathbb{R}^{n\times n}$ and the vector of external forces acting on the arm $\mathbf{n} \in \mathbb{R}^n$ which includes gravitational forces [6]. The zero position of the arm $\boldsymbol{\theta} = 0$ is set to be when the arm is pointing straight up as shown in Figure 4.1. The model depends on the current position and velocity of the joints, which are denoted as $\boldsymbol{\theta}$ and $\dot{\boldsymbol{\theta}}$ respectively. Parameters required to calculate the model are found on the official youBot homepage[1]. The following subsections describe the single components of the dynamical model in more detail.

[1] `http://www.youbot-store.com/youbot-developers/software/simulation/kuka-youbot-kinematics-dynamics-and-3d-model?c=44`

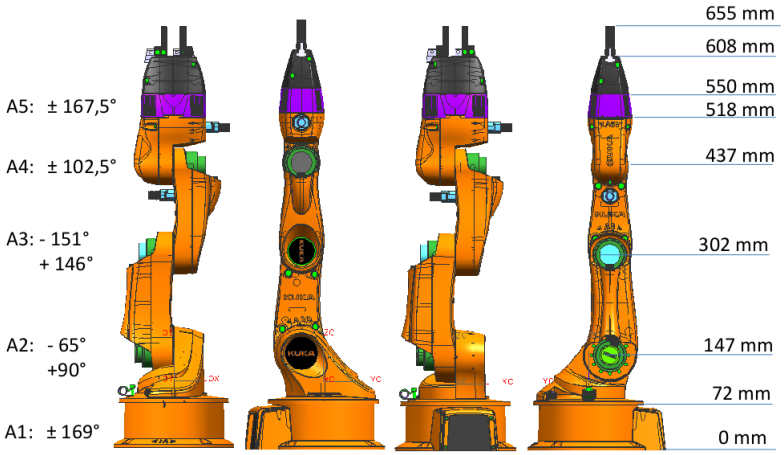


Figure 4.1: Youbot arm with height measurements and angle limits.

4.1.1 Manipulator inertia matrix

To calculate the manipulator inertia matrix $\mathbf{M}$, the link inertia matrix

$$\mathbf{M}_i = \left[\begin{array}{ccc|ccc} m_i & 0 & 0 & & & \\ 0 & m_i & 0 & & \mathbf{0} & \\ 0 & 0 & m_i & & & \\ \hline & & & I_{i,x} & 0 & 0 \\ & \mathbf{0} & & 0 & I_{i,y} & 0 \\ & & & 0 & 0 & I_{i,z} \end{array}\right] \tag{4.2}$$

has to be known for every link i. The matrices $\mathbf{M}_i$ depend on its mass m_i and its moment of inertia I_i. All the links are assumed to be symmetric and hence have a diagonal moment of inertia matrix. In addition to the link inertia matrix, we need to compute the body Jacobian corresponding to each link frame. Interested readers can find the derivation of the body Jacobian matrix in the book "A Mathematical Introduction to Robotic Manipulation"[6] on pages 115-117. The manipulator inertia matrix can then be calculated as

$$\mathbf{M}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \mathbf{J}_i^T(\boldsymbol{\theta}) \mathbf{M}_i \mathbf{J}_i(\boldsymbol{\theta}). \tag{4.3}$$

It only depends on the joint positions of the manipulator and is a symmetric and positive definite matrix.

4.1.2 Coriolis matrix

The Coriolis matrix for the manipulator $\mathbf{C}$ gives the Coriolis and centrifugal force terms in the equation of motion [6]. It depends on both the joint positions and the joint velocities of the manipulator. We can calculate the elements of $\mathbf{C}$ directly from the manipulator inertia matrix as

$$\mathbf{C}_{ij}(\boldsymbol{\theta}, \dot{\boldsymbol{\theta}}) = \frac{1}{2} \sum_{k=1}^{n} \left(\frac{\partial \mathbf{M}_{ij}}{\partial \boldsymbol{\theta}_k} + \frac{\partial \mathbf{M}_{ik}}{\partial \boldsymbol{\theta}_j} - \frac{\partial \mathbf{M}_{kj}}{\partial \boldsymbol{\theta}_i} \right) \dot{\boldsymbol{\theta}}_k, \tag{4.4}$$

with i and j denoting the row and column of the matrix and k the joint number.

4.1.3 Force vector

For the force vector $\mathbf{n}$, all the external forces acting on the manipulator have to be considered. In this project we assume that the only forces acting on the manipulator come from the gravitational pull. The gravitational forces can then be computed by calculating the potential energy

$$V(\boldsymbol{\theta}) = \sum_{i=1}^{n} m_i g h_i(\boldsymbol{\theta}) \tag{4.5}$$

of the manipulator. Where g is the gravitational acceleration and h_i the height of the center of gravity of link i. From the potential energy we can calculate the gravitational forces acting on the manipulator as

$$\mathbf{n}(\boldsymbol{\theta}) = \frac{\partial V}{\partial \boldsymbol{\theta}}. \tag{4.6}$$

In addition to gravitational forces, frictional and damping forces could be added to $\mathbf{n}$ as well but they are neglected in the dynamical model of the youBot.

4.2 Computed-torque control

With the derived dynamical model of the robot, the required joint torques for given joint positions, velocities, and accelerations can be calculated according to Equation 4.1. Applying these torques to the manipulator results in an open-loop torque controller. For a perfect model with corresponding initial conditions, the manipulator will match the desired position and velocity and thus correctly track a trajectory. In order to make tracking more robust, a state feedback is introduced. The feedback is necessary to cope with model uncertainties, disturbances, and wrong initial conditions. This can be written as the computed-torque control law

$$\boxed{\boldsymbol{\tau} = \mathbf{M}(\boldsymbol{\theta})(\ddot{\boldsymbol{\theta}}_d - \mathbf{K}_v\dot{\mathbf{e}} - \mathbf{K}_p\mathbf{e}) + \mathbf{C}(\boldsymbol{\theta}, \dot{\boldsymbol{\theta}})\dot{\boldsymbol{\theta}}_d + \mathbf{n}(\boldsymbol{\theta})}, \tag{4.7}$$

where $\boldsymbol{\theta}_d$ denotes the desired joint positions and $\mathbf{e} = \boldsymbol{\theta} - \boldsymbol{\theta}_d$ the error in joint positions. The matrices $\mathbf{K}_p$ and $\mathbf{K}_v$ consist of the proportional and differential feedback gain of each link respectively. Substituting Equation 4.7 into 4.1 yields the error dynamics of the control law.

$$\mathbf{M}(\boldsymbol{\theta})(\ddot{\mathbf{e}} + \mathbf{K}_v\dot{\mathbf{e}} + \mathbf{K}_p\mathbf{e}) = 0 \tag{4.8}$$

As $\mathbf{M}$ is always positive definite this can be simplified to

$$\ddot{\mathbf{e}} + \mathbf{K}_v\dot{\mathbf{e}} + \mathbf{K}_p\mathbf{e} = 0 \tag{4.9}$$

From Equation 4.9 it is derived that by choosing $\mathbf{K}_p$ and $\mathbf{K}_v$ positive definite and symmetric, the control law (Equation 4.7) results in an exponentially stable system. In addition, we also choose the gain matrices to be diagonal only, which allows us to split the system up into five separate systems. That is why we can tune the feedback gain for each joint independently.

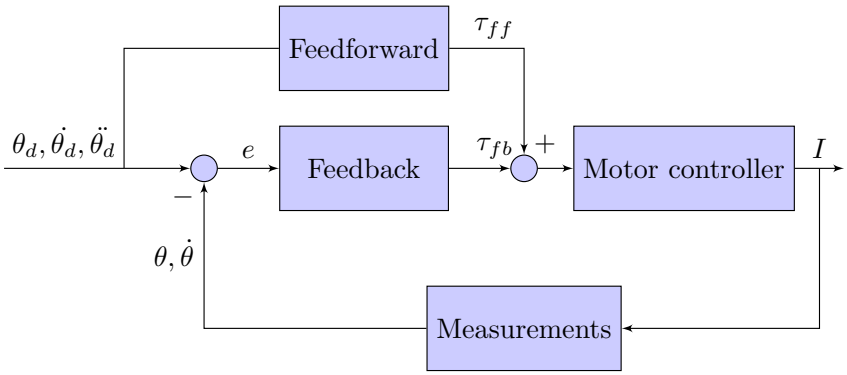


Figure 4.2: Control scheme for the computed-torque controller. Feedforward and feedback form the controller which sends the torque to the motor controllers. The torques are then translated into currents and we receive measurements about joint positions and joint velocities back.

Further analysis of Equation 4.7 leads to the separation of the control law in a feedforward and a feedback term.

$$\boldsymbol{\tau} = \underbrace{\mathbf{M}(\boldsymbol{\theta})\ddot{\boldsymbol{\theta}}_d + \mathbf{C}(\boldsymbol{\theta}, \dot{\boldsymbol{\theta}})\dot{\boldsymbol{\theta}}_d + \mathbf{n}(\boldsymbol{\theta})}_{\boldsymbol{\tau}_{ff}} + \underbrace{\mathbf{M}(\boldsymbol{\theta})(-\mathbf{K}_p\mathbf{e} - \mathbf{K}_v\dot{\mathbf{e}})}_{\boldsymbol{\tau}_{fb}} \tag{4.10}$$

This allows us to set up the control scheme illustrated in Figure 4.2.
The manipulator inertia matrix $\mathbf{M}$ in the computation of the feedback torques provides a positional scaling of the torque. Depending on the current joint positions, a small change in torque can have a huge or a very small impact on the movement of the joints. This property is very important to the feedback controller as we do not want to correct the feedforward torque by too much or too little. Since we want to keep the system separable, we choose to only use the diagonal elements of $\mathbf{M}$ in the feedback term. This way, the system with the joint position dependent gain matrices $\mathbf{M}(\boldsymbol{\theta})\mathbf{K}_p$ and $\mathbf{M}(\boldsymbol{\theta})\mathbf{K}_v$ can still be split up in five separate systems.

4.2.1 Tuning of the feedback controller gain matrices

Tuning the gain matrices $\mathbf{K}_p$ and $\mathbf{K}_v$ is done by analysing the step responses for each joint separately and increasing or decreasing the gain accordingly [9]. An increase in the proportional gains $\mathbf{K}_p$ results in a faster rise time but also in a larger overshoot and degrading stability of the system. As we do not want gains high enough to cause an unstable system, it is better to start with a lower gain and gradually increase it, until the step response is satisfactory. Figure 4.3 shows such a step response. Fine tuning is done afterwards by analysing how closely a joint space trajectory is followed and adjusting the controller gain accordingly. A starting point for the tuning of the gains is found based on the maximum applicable torques and maximised manipulator inertia matrix $\mathbf{M}$. By

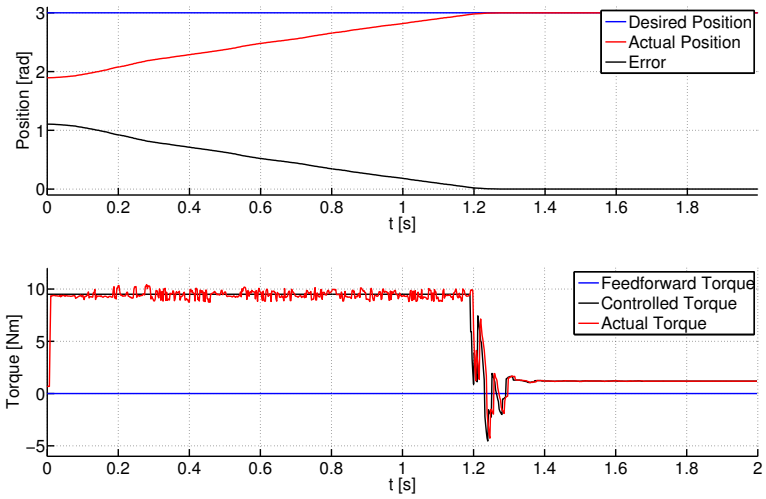


Figure 4.3: Response of a joint position step applied to joint 1. The maximum torque in this joint is limited to 9.5 N m.

following the equation

$$\boldsymbol{\tau}_{max} = \mathbf{M}_{max}\mathbf{K}_p e_{max}, \tag{4.11}$$

where $\mathbf{e}_{max}$ is the maximum possible joint position error, depending on the position that maximises $\mathbf{M}$. Each joint can then be approximated by the second order transfer function

$$H(s) = \frac{1}{s^2 + k_{i,v}s + k_{i,p}}. \tag{4.12}$$

This transfer function shows that by choosing the elements of $\mathbf{K}_v$ as $2\sqrt{k_{i,p}}$ the system becomes critically damped.
Note that $\mathbf{e}_{max}$ can reach values of up to 5.8 rad depending on the joint. As we want accurate trajectory tracking with very low position errors, the gain has to be set to large values. This means that for large position errors the torque controller would output joint torque values that are way above the maximum applicable joint torque. So in order to prevent damage to the motors, the controller output should be artificially limited to the maximum applicable joint torque.

4.3 Motor controllers

Each joint motor of the KUKA youBot manipulator is controlled by a Trinamic TMCM-1610-KR controller board. This controller board regulates the applied current based on the command it received. Figure 4.4 shows the cascaded PID regulation loop used to do so. Commands received can either be a target

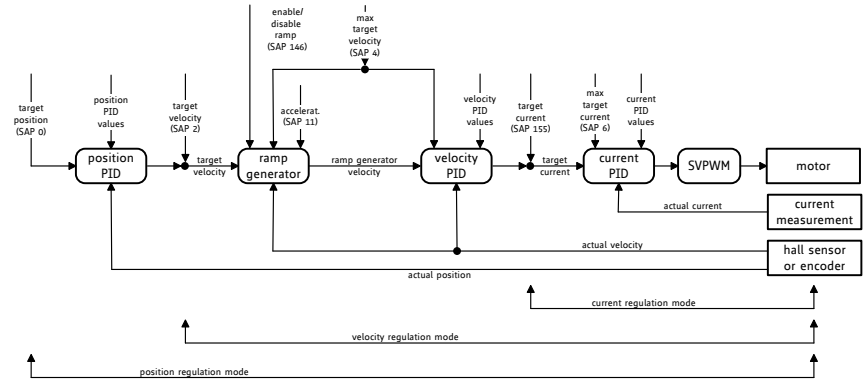


Figure 4.4: Cascaded PID regulation of the motor controller board. The three possible commands are target position, target velocity and target current.

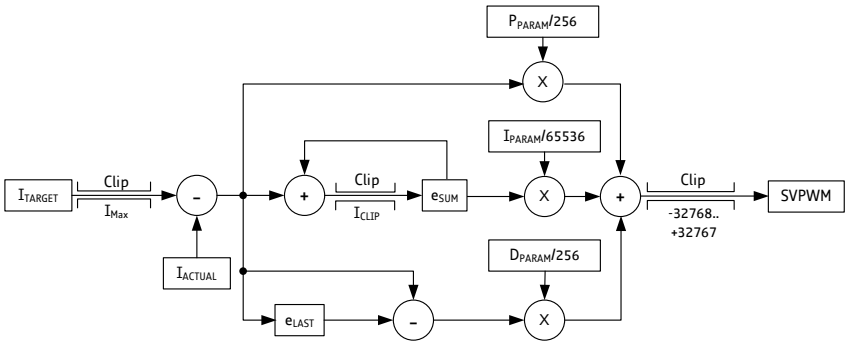


Figure 4.5: Current PID regulation mode of the motor controller.

position, velocity, or current. The torque commands from the torque controller, are translated into target currents using a motor model. So for the torque controller only the current regulation mode, which consists of a single PID loop (Figure 4.5), is of importance.

The torque controller relies on an excellent reaction to commanded set-points. In other words, we require very short rise and settling times as well as little to no overshoot. This was not the case with the factory presets. Especially the rise time was far too long which can be seen in the step response in Figure 4.6. So to get the torque controller to work properly, the current controllers need to be tuned properly. Tuning of the gains is done by analysing step responses of the current controllers. As a moving joint introduces a lot of noise in the current measurement (see Figure 4.6), we prevented the joints from moving while applying a step input on the current. In addition, due to the noise in the current measurement of a moving joint, it is already derived that introducing a differential gain would prove counter productive and might even cause the controller to become unstable. That is why we only use a PI controller.

We tuned the PI controller manually by starting with a low proportional gain and no integral gain. The proportional gain is then increased until the current

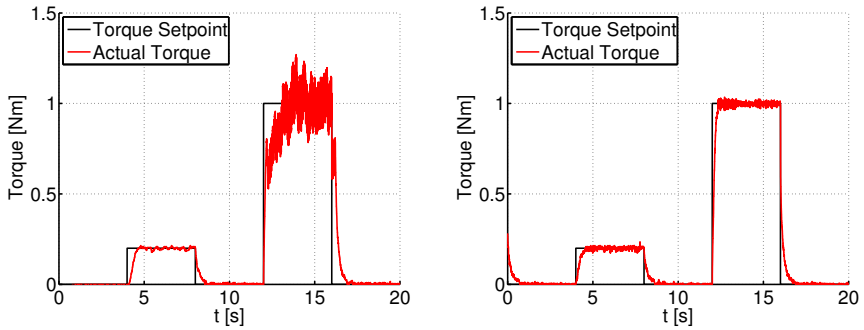


Figure 4.6: Step response of the current PID controller of joint 1 when the joint is moving (left) and when it is blocked (right).

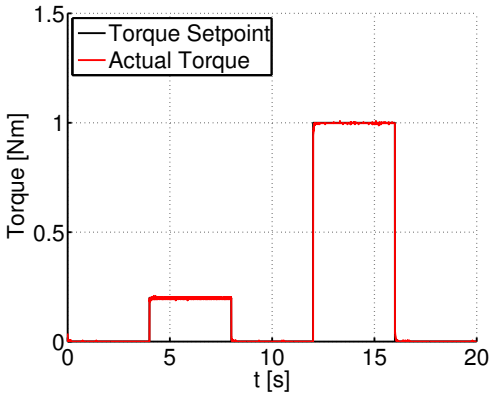


Figure 4.7: Step response of the current PID controller of joint 1 after tuning its gains.

response starts oscillating. It is then roughly halved and the integral gain is increased until there is no steady state offset. Figure 4.7 shows the step response of the current PID regulator after this tuning step. Comparing it to the step response in Figure 4.6, it is seen that the rise time is a lot faster with no overshoot and instant settling. After tuning the current PID loop, the same procedure was done for the position and velocity controller gains to ensure the desired functionality of these control methods. Table 4.1 gives a comparison of the existing and the tuned controller gains.

4.4 Trajectories

We consider discrete time trajectories consisting of a point for every time-step. The time-step is depending on which frequency the torque controller is running at. Every point needs to have a value for desired position, velocity and acceleration for every joint. Trajectories are generated in the Cartesian space for the desired end-effector pose. These Cartesian poses are then converted to joint positions using the inverse kinematic chain of the youBot arm [8]. Joint velocities

Joint			1	2	3	4	5
existing	current	K_p	25	25	25	25	25
		K_i	60	60	60	60	60
		K_d	0	0	0	0	0
	velocity	K_p	1000	1000	1000	1000	1000
		K_i	1000	1000	1000	1000	1000
		K_d	0	0	0	0	0
	position	K_p	2000	2000	2000	2000	2000
		K_i	0	0	0	0	0
		K_d	0	0	0	0	0
tuned	current	K_p	1200	1200	1200	2000	4000
		K_i	3000	3000	3000	4000	4000
		K_d	0	0	0	0	0
	velocity	K_p	300	300	300	200	120
		K_i	600	600	600	800	100
		K_d	0	0	0	0	0
	position	K_p	1000	1000	1000	1000	1000
		K_i	0	0	0	0	0
		K_d	0	0	0	0	0

Table 4.1: Comparison of existing and tuned controller gains.

and accelerations are calculated using the discrete time derivatives. While no feedback exists for the acceleration, the desired acceleration is still important in the calculation of the feedforward torques. The aim of the designed torque controller is to closely track end-effector trajectories in Cartesian space. This is necessary to allow accurate and precise grasping of objects. For example for easy and quick grasping, we assume that the end effector is placed directly above the object. Then the torque controller should lower the end effector in a straight line to encompass the object. Figure 4.8 shows the end-effector trajectory as well as the joint positions for such a straight line with velocity and acceleration profiles seen in Figure 4.9.

4.5 Self-tuning fuzzy logic controller

In addition to the standard computed-torque controller, a self-tuning controller via fuzzy logic was also tested. This is useful in real applications of a computed-torque controller where unmodeled disturbances come into effect [4]. The idea behind this is quite simple and builds on Equation 4.11. As an increase of the gains will decrease the steady state error [9] the aim is to have $||\mathbf{K}_p||$ as big as possible while retaining system stability. The maximum gain values without generating torque values bigger than the maximum allowed torque, is dependent on the tracking error $\mathbf{e}$. The bigger the error, the smaller the gains have to be to still generate applicable torques. So the idea is to have the gain matrices tune themselves based on the current tracking error. We chose fuzzy logic to introduce this self tuning property. It allows us to define fuzzy rules such as *"If the position error of joint 1 is very small then the corresponding gain in* $\mathbf{M}$

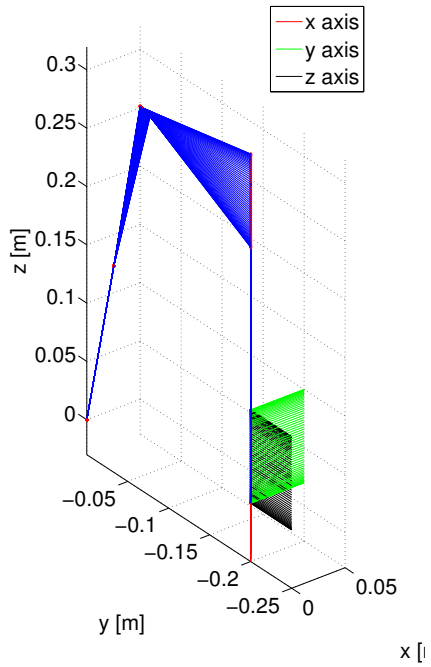


Figure 4.8: Example trajectory in Cartesian space. The x, y, and z axis symbolise the gripper orientation. The red points are the different joints.

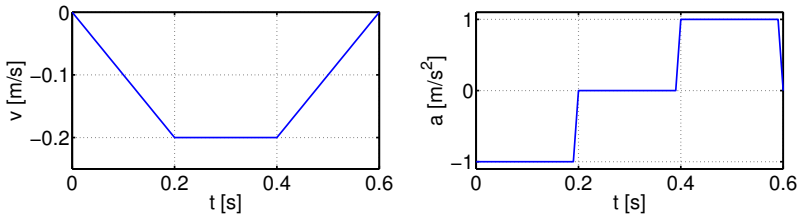


Figure 4.9: End-effector velocity and acceleration trajectories corresponding to Figure 4.8.

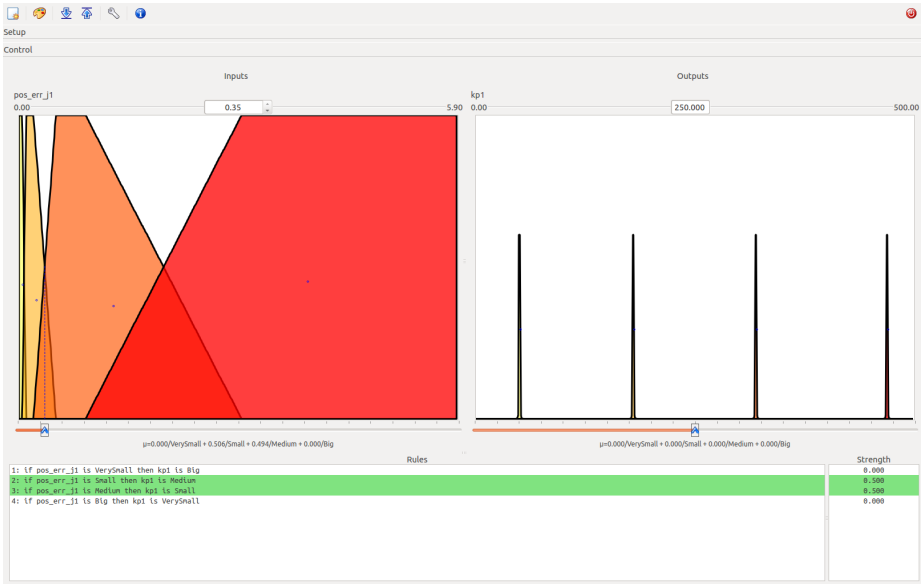


Figure 4.10: Fuzzy controller definitions for joint 1. On the left four functions describing keywords for the input variable can be seen. The right part shows the keyword functions for the output variable. From this the controller gain is calculated.

should be big". Keywords such as *very small* or *big* are then defined by various functions (see Figure 4.10). We can then collect all these fuzzy rules to make up the feedback gain matrices. As the fuzzy logic controller did not improve the control performance in case of the youBot, this method was discarded again.

Chapter 5

Object detection

The detection of objects we can grasp is done using the PCL to analyse a point cloud. The scene is assumed to be sparse and uncluttered. This means we have a low total object count with no obstructions or background noise. This chapter describes the algorithms used for detecting objects in such a scene. Figure 5.1 provides an overview over the whole object detection algorithm.

5.1 Downsampling of the Point Cloud

The output point cloud of the ASUS Xtion has a point for each pixel. At VGA resolution this results in 307200 points. Analysing this point cloud directly would be very computationally expensive. In a first step, we downsample the cloud. This is done using a voxel grid filter[1]. A voxel is basically a 3D box. So a voxel grid can be seen as a set of 3D boxes attached to each other to cover the space. Now for all points in a voxel, their centroid point is calculated and saved in a new point cloud. This results in a point cloud with far less points that is still approximating the underlying surface very nicely. For example, a cubic voxel with an edge length of 5 mm leads to a point cloud consisting of about 65000 points which is 5 times less than the original.

5.2 Removal of the planes

As the objects are expected to lie on a flat surface such as the ground, the next step in the detection algorithm is to remove all the flat surfaces. This is done using a plane segmentation algorithm[2]. The PCL provides an SAC model for a plane and RANSAC is then used to find all the inliers of the model. A distance threshold of 1 cm is set as a limit to determine the inliers. For the detected plane the estimated plane parameters are returned which are used to differentiate between floors and walls. The algorithm then removes all the found planes from the original point cloud and stores the remaining points in a new cloud. While the randomness of RANSAC does not guarantee an optimal

[1] `http://pointclouds.org/documentation/tutorials/voxel_grid.php#voxelgrid`

[2] `http://pointclouds.org/documentation/tutorials/planar_segmentation.php#planar-segmentation`

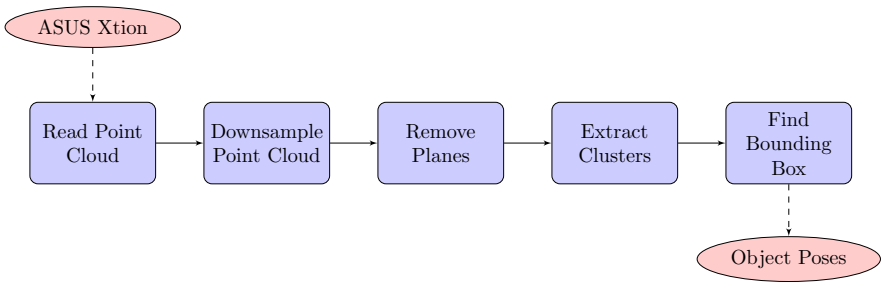


Figure 5.1: Flowchart of the detection algorithm

solution, its robustness and short execution time are far more important for the detection algorithm which makes it a great choice.

5.3 Cluster Extracting

The remainder of the original point cloud now only consists of points that could be objects. In order to check if there are any objects in the scene, the points are grouped into clusters[3]. A cluster contains only points that fulfil some requirement of closeness, meaning the points should belong to the same object. This is done by setting up a k-d tree object and using a nearest-neighbour classification to find points that belong to the same cluster. The k-d tree sets up the points in a way that optimises lookups in the k-dimensional space and supports nearest-neighbour queries [2]. The nearest-neighbour algorithm assigns a point the same value as its nearest-neighbour, assuming the points are within a certain distance of its neighbour. So points that lie close to each other will be grouped in the same cluster. In this algorithm points can be no further than 1 cm of each other. Graspable objects have to be fairly small due to the limited width the gripper which has a maximum opening of 6 cm. This allows us to impose an additional limit on the cluster size. Only clusters that have at least 40 points but less than 500 points get saved.

5.4 Bounding boxes

The resulting clusters now have to be further analysed to filter out any unwanted clusters and receive information about the pose of the remaining clusters. We assume that graspable objects are boxes with an edge length of at least 2 cm and a maximum width of 10 cm, height and depth of 5 cm. To get the size of the cluster as well as its pose we can apply an algorithm to find a bounding box for the cluster. It is based on PCA and the projection on the axes of the coordinate frame. From the projection we can calculate the width, length and depth of the cluster, while PCA returns the translation and orientation of the cluster. The resulting orientation returns inaccurate values for roll and pitch. This stems from the asymmetric nature of the cluster, with only three faces of

[3] `http://pointclouds.org/documentation/tutorials/cluster_extraction.php#cluster-extraction`

the box visible in the best case scenario. The calculated yaw is accurate as it is dependent on the two faces with most visible points. As we assume the objects to lie flat on the floor, roll and pitch can be manually set to zero. If a cluster's size fulfils the object requirements, its pose is stored in a vector of poses. This is the final step in the object detection algorithm. In Figure 5.2 an example point cloud can be seen with the objects highlighted in green.

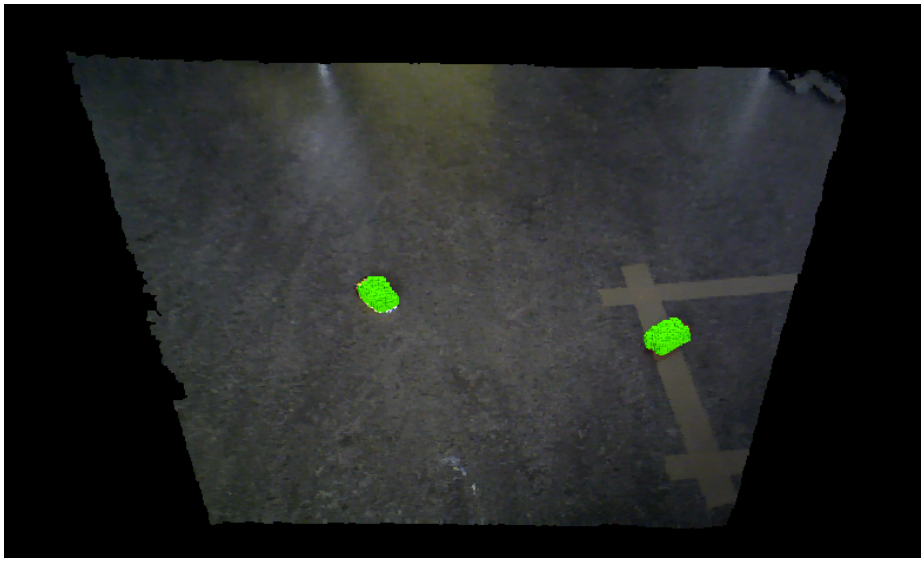


Figure 5.2: Point Cloud with detected objects highlighted in green

Chapter 6

Experimental results

In this chapter, the performances of both the torque controller and the object detection algorithm are analysed.

6.1 Computed-torque control on the KUKA youBot

To evaluate the torque controller, we look at different trajectories. The first trajectory is the one which motivated the project, namely grasping an object by lowering the end effector from a position directly above the object. Then we look at a much longer trajectory, which is used for grasp while driving, where the arm gets dragged parallel to the base while lowering the manipulator and then lifting it back up. Finally, we analyse tracking of a complicated trajectory in form of a circle where all the joints have to be moved simultaneously and the controller is operating near the joint position limits.
Figure 6.1 shows the actual end-effector trajectory driven by the torque controller, when attempting to grasp objects directly from above. The trajectory was generated using a maximum end-effector velocity of $0.05\,\frac{\mathrm{m}}{\mathrm{s}}$ and a maximum end-effector acceleration of $0.5\,\frac{\mathrm{m}}{\mathrm{s}^2}$. It is completed in 1.7 s. Comparing this to the results in Figure 1.1 and Figure 1.2, it is clear that the performance of the torque controller is a lot better than the existing methods. It has a similar accuracy as lowering the arm in 1 mm steps but it completes the task 10 times faster. As the time required to follow a trajectory is crucial for a large range of dynamic tasks, the fact that the torque controller is that much faster than the 1 mm step grasping cannot be emphasised enough. Comparing the absolute tracking error of the torque controller to the existing methods, illustrated in Figure 6.3, shows that the torque controller stays within a few millimetres of the desired trajectory. Only the 1 mm step method has a smaller mean error but this method is not practical due to the long runtime. The absolute error is the 2 norm of the errors in x, y, and z direction and is illustrated for the torque controller in Figure 6.2.
Looking at a longer trajectory, as seen in Figure 6.4, which was again generated with a maximum end-effector velocity of $0.05\,\frac{\mathrm{m}}{\mathrm{s}}$ and a maximum end-effector acceleration of $0.5\,\frac{\mathrm{m}}{\mathrm{s}^2}$, we can see that much longer trajectories can still be tracked with good accuracy. Figure 6.5 shows the absolute error in position tracking over the 7 s it took to complete the trajectory. The error peaks at

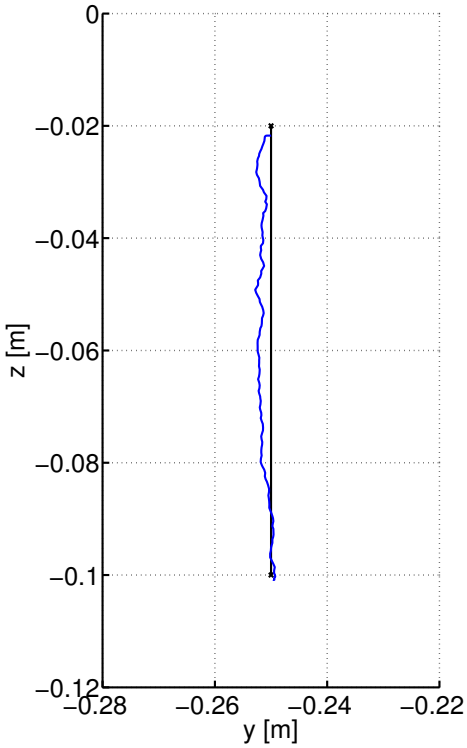


Figure 6.1: The actual end-effector position (blue) during the grasping trajectory compared to the desired trajectory (black). The trajectory was completed in 1.7 s.

4.2 mm with the average being 2.2 mm.

Figure 6.6 shows how the error in tracking of the grasp while driving trajectory changes depending on the maximum velocity the trajectory was generated with. As expected, the faster the trajectory has to be completed, the bigger the error in tracking. Unexpectedly, for very slow velocities, there is a large number of points with a high end-effector position error which reduces the quality of the tracking. This is due to friction and the self-locking properties of the youBot joints. The lowest error was recorded at a maximum end-effector velocity of $0.03\,\frac{\mathrm{m}}{\mathrm{s}}$.

So far we only considered straight end-effector trajectories which can be used for various grasping tasks. While these paths are important, they are also quite simple. To really put the torque controller to the test, we generated a more complex trajectory in form of a circle in Cartesian space. In order to follow a circle, all joints of the youBot have to be moved in both directions during the whole process and all joints are moving simultaneously. In addition, the torque controller is also operating near the joint position limits. Figure 6.7 shows the desired trajectory together with the actual end-effector position of the process which took 20 s to complete. Figure 6.8 shows the corresponding

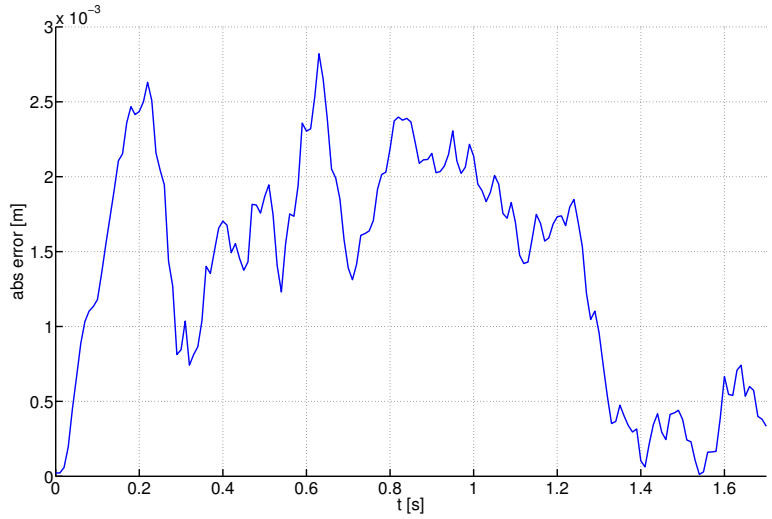


Figure 6.2: Absolute error during the grasping trajectory.

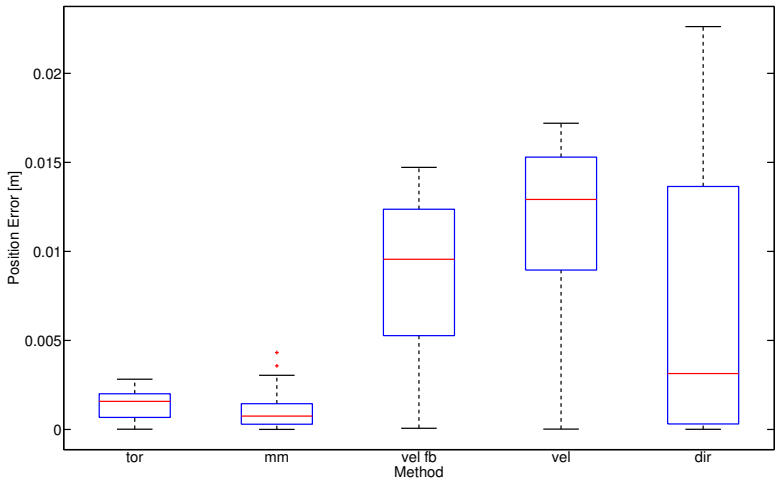


Figure 6.3: Error of the different control methods (torque control, 1 mm steps, velocity with feedback, velocity, direct) compared with each other.

absolute error over the time period. During the first half of the trajectory, where the arm is lowered, tracking is accurate. As soon as the arm is lifted up again, an offset in z direction is introduced by the torque controller. The offset originates from physical limitations of the youBot manipulator and its inherent problems of lifting arm joints in certain configurations in the torque control mode (see Chapter 7.1). This problem is intensified by the self-locking joints of the youBot. At joint velocities close to 0 $\frac{\mathrm{m}}{\mathrm{s}}$ the joint stops completely and needs

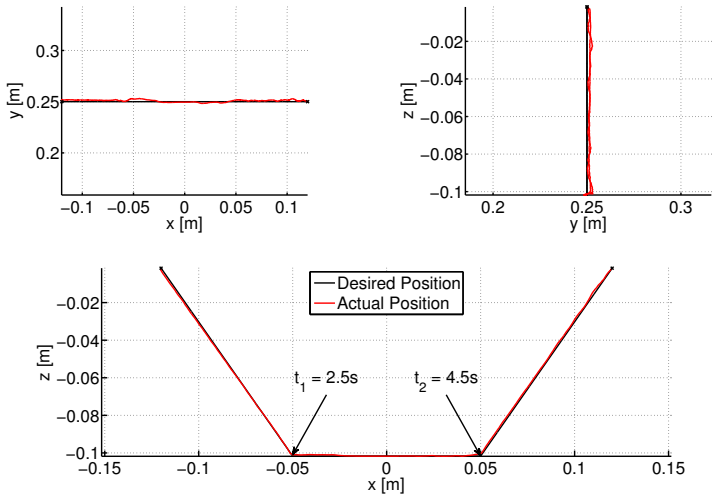


Figure 6.4: Tracking of a trajectory for grasp while driving. $\Delta t = 7\,\mathrm{s}$.

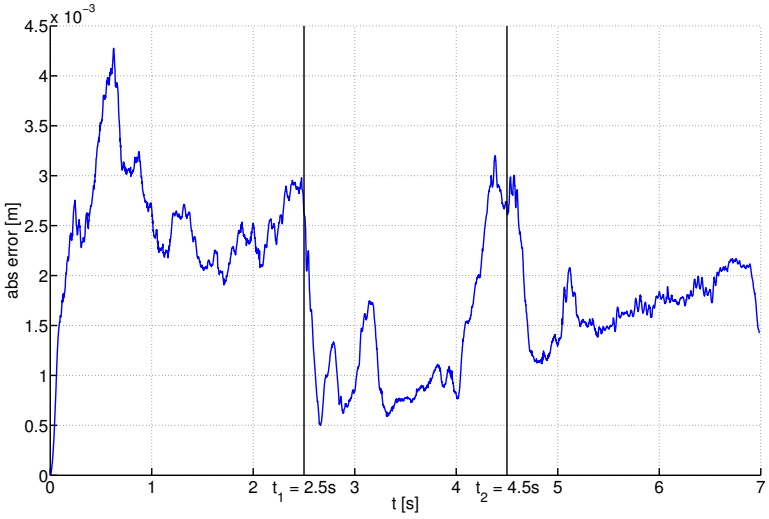


Figure 6.5: Absolute error while tracking the grasp while driving trajectory.

a high torque to get moved again. This effect is not included in the dynamical model of the youBot and has a negative consequence on the performance of the torque controller.

6.2 Object detection and grasping

For autonomous grasping, the ASUS Xtion sensor used for object detection is mounted on the youBot. To evaluate the performance of the object detection

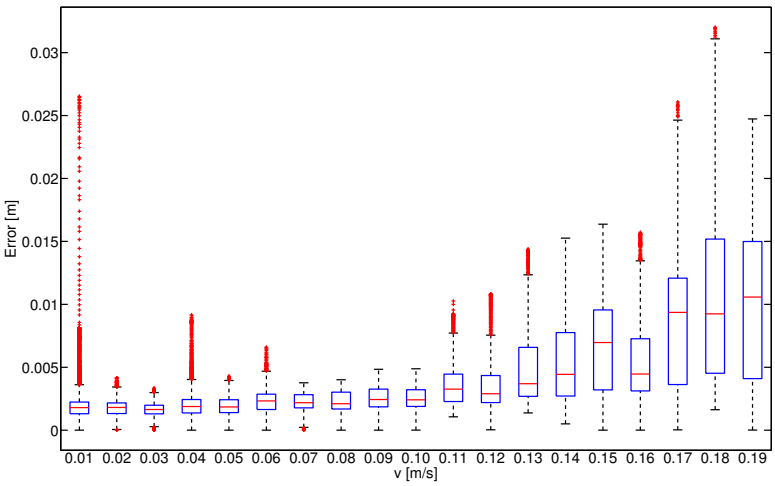


Figure 6.6: Errors in end-effector position of the grasp while driving trajectory for different velocities.

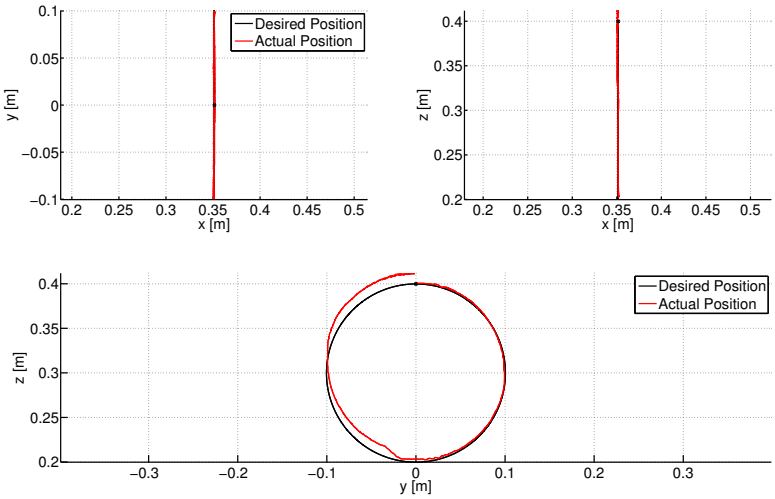
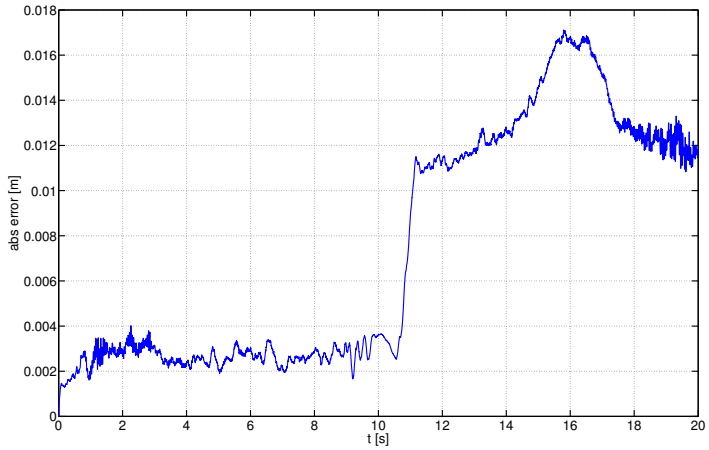


Figure 6.7: Tracking of a circular trajectory

algorithm, we placed an object at a known position and moved the youBot around it while simultaneously tracking the position of the youBot using a motion capture system. We then assume that the youBot was standing still, while the object was placed at different positions. Then the detected positions are compared to the actual positions which is illustrated in Figure 6.9. In addition, the difference in detection of the algorithm and the motion capture system can be split up in the error in x and y direction and the rotation angle.



m

Figure 6.8: Absolute error while tracking a circle, it is shown that at $t = 10\,\mathrm{s}$ an offset gets introduced which comes from problems with lifting the arm properly.

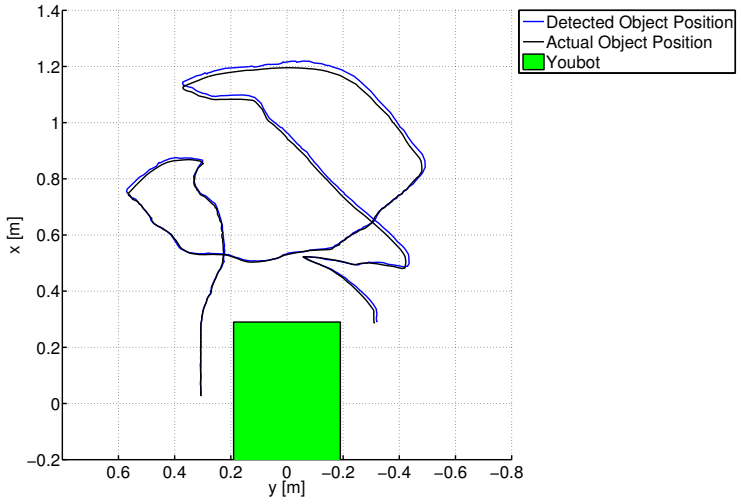


Figure 6.9: Detected objects compared to the actual object positions

These plots are shown in Figure 6.10.
Finally, the absolute error in detection, is compared to the distance of the object to the youBot (see Figure 6.11). The large errors of up to 2 cm in each direction with up to 12 degrees in the rotation angle, is tracked back to various reasons. Firstly, the accuracy depends on the actual position of the Kinect sensor. As that position cannot be measured accurately, the accuracy of the algorithm is suffering. Another reason is that the precision of the sensor is decreasing with increasing distance and usually ranges between 1 mm and 1 cm[1]. The accuracy

[1] `http://wiki.ros.org/openni_kinect/kinect_accuracy`

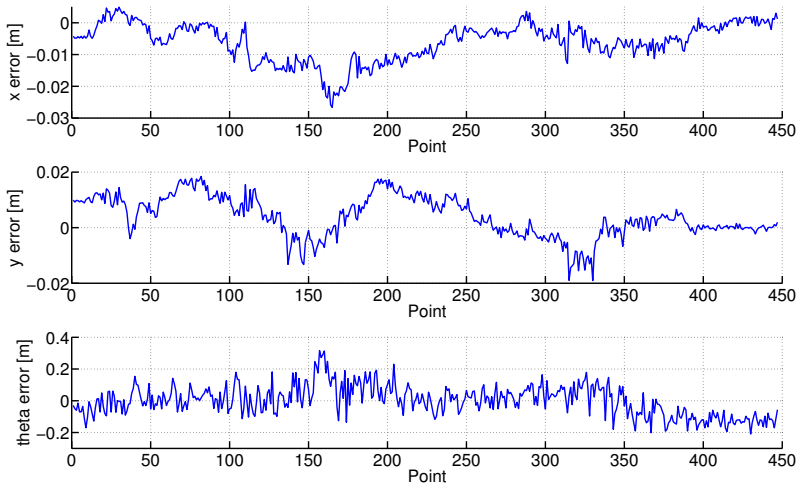


Figure 6.10: Error of the object detection in x and y direction and in the rotation angle.

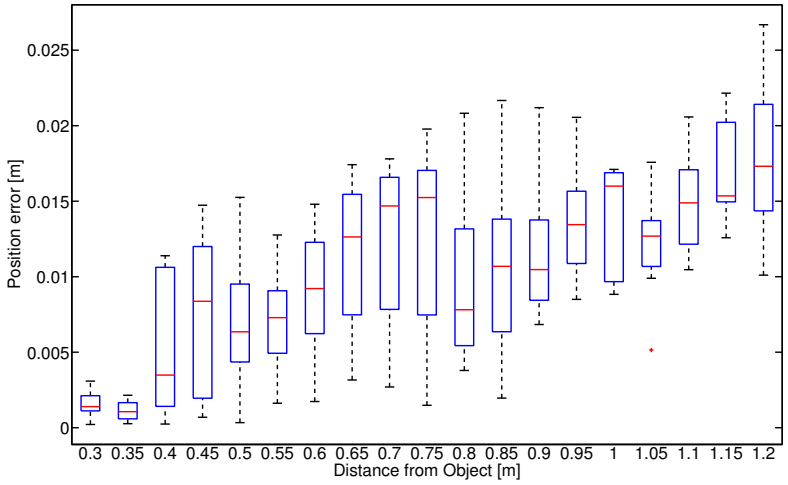


Figure 6.11: Absolute error, as the 2 norm of the errors in x and y direction, of the object detection depending on the distance from the object

of the sensor also depends on the lighting conditions as the infra-red sensor can be disturbed by ambient light. Even though the detected object positions are not very accurate, we were still able to successfully perform autonomous grasping and grasp while driving (see Chapter 7.2) using this detection method together with the developed torque controller.

Chapter 7

Discussion

7.1 Torque controller

The torque controller is split up in a feedforward and a feedback term. From the dynamical model of the youBot we can directly calculate the feedforward term. With the feedback term we then correct this calculated torque based on the error in joint position and joint velocity. After tuning the PID gains of the motor current controllers and the feedback gain matrices the torque controller is able to track trajectories.

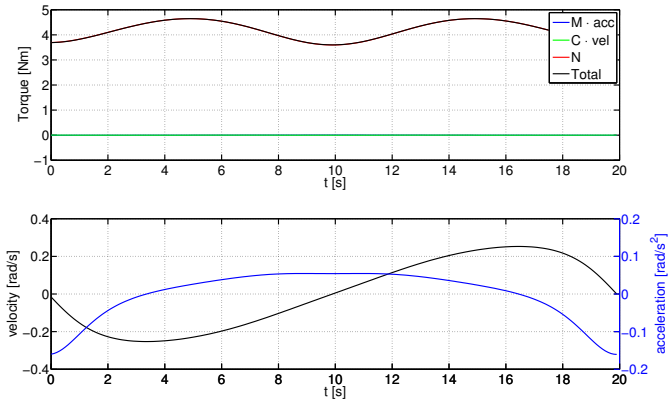


Figure 7.1: Calculated torques with velocity and acceleration plots. It can be seen that only N, the part compensating the gravity adds to the total required torque and the dynamic parts $\mathbf{M} \cdot acc$ and $\mathbf{C} \cdot vel$ stay zero.

The performance of the torque controller varies depending on the trajectory that is tracked. If the desired path is well within torque limits, the tracking only deviates slightly from the desired trajectory. These errors in tracking are small enough, even at fast speeds, to complete several tasks that require a dynamic and efficiently controlled manipulator. However, there are limitations which mostly stem from the model parameters the controller is based on. Firstly, the current dynamical model parameters are wrong, which is illustrated in Figure

7.1. The calculated torques are almost independent of joint velocities and joint accelerations. This results in a huge offset in calculated torque and the torque that is actually required to track a trajectory. Secondly, the dynamical model is incomplete. Friction and the self-locking properties of the youBot joints are not considered. This shifts all the weight of the torque controller on the feedback loop, meaning that without it, the arm is not moving at all. The resulting large offsets require a more aggressively tuned feedback controller, which in turn can become critically stable in certain situations, which leads to bad performance.

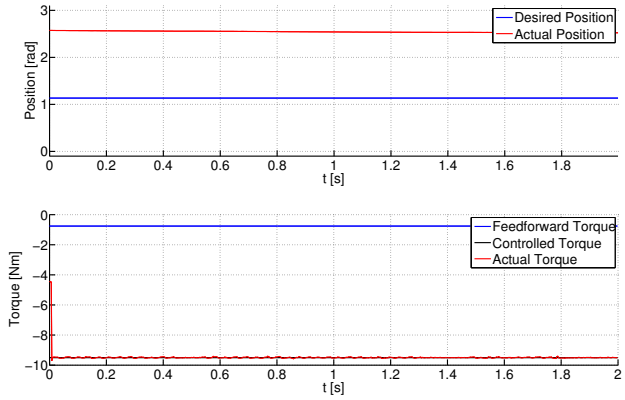


Figure 7.2: Joint position and joint torque for arm joint 2 after a commanded step input with fully stretched out arm. The arm cannot be lifted because the imposed torque limit of 9.5 N m is too small.

Another reason why the performance of the torque controller is suffering at times is because of the maximum torque on the joints imposed by the manufacturer. They are simply too small and do not even allow the manipulator to move a joint to a different position in certain configurations even though those movements are possible using the direct position control or velocity control methods. Figure 7.2 shows the position and torque on joint 2 after it was commanded to lift the fully stretched out arm. It shows that even though the maximum allowed torque is commanded, the arm does not move in the desired direction.

7.2 Object detection and grasping

Objects are detected from the point cloud captured by an ASUS Xtion PRO sensor. The point cloud is then downsampled using a voxel grid filter. After RANSAC based plane removal, the remaining points are sorted into clusters using a k-d tree. For each of these clusters, we then find the minimum bounding box that includes all the points. If the size of the box is in the range we expect our objects to be, the cluster is then classified as an object and its pose is saved. While the object detection algorithm detects objects accurately enough to grasp them autonomously, it is also limited in its usefulness. In our experiments, autonomous grasping using the object detection algorithm and the developed torque controller worked 90 % of all the attempts. For objects close to the base,

the detection will not work. In this case the Kinect sensor is obstructed by the base of the youBot. Additionally, during the grasp while driving process, the camera is obstructed by the arm of the youBot. Therefore, closed-loop grasping of objects is impossible. As the open-loop base movement of the grasp while driving experiment is a lot longer than for the autonomous grasping, the success rate was much lower. Without accurate position updates, we relied entirely on the odometry data of the youBot. As this data is drifting and is not very accurate, grasp while driving only worked 50 % of all the attempts.

7.3 Future Work

To improve the tracking properties of the torque controller for complex trajectories and close to the joint limits, it is necessary that the model discrepancies are resolved. The model parameters from the manufacturer have to be verified and corrected where necessary. Also, the new model should include friction and compensation for the self-locking effect of the joints.
If robust grasp while driving is required, the object detection algorithm has to be extended. It would be best to only use the Kinect sensor to initially estimate the object position. The accurate position could then be confirmed by the use of a different sensor once the youBot is close to the object. This could be done by mounting a simple color camera on the youBot arm. From its image, the accurate object position can then be extracted and the arm position corrected accordingly.

Appendix A

Orocos

The Open Robot Control Software[1] (Orocos) is an environment developed for robotics applications. Orocos is built on a real time messaging system which is a necessity for some applications. It offers the possibility to send ROS messages which enables the two systems to be connected. Orocos is a peer based system which allows the passing of parameters and messages between connected peers. The downside of Orocos is that it is a lot less common and offers fewer tools than ROS. It is also not as transparent, which makes data logging more difficult. There is an unofficial Orocos driver for the KUKA youBot, which allows sending of position, velocity and torque commands. However, unlike the ROS component, the Orocos driver is not official and methods to write the commands to the youBot differ greatly. After initially using the Orocos driver the switch to ROS was made when twice in a short period of time joint controllers were damaged due to over current. It was never proven that the controller was damaged due to the use of Orocos or not, but the problem ceased to exist after the switch to ROS.

[1] `http://www.orocos.org/`

Appendix B

Coordinate frames

As the absolute joint positions, which the youBot is using, are not suited for kinematic or dynamic calculation, they have to be converted to a better suited system. For this, two different coordinate system were introduced independently of each other. The problem stems from the fact that the axis of rotation for joint 3 is opposite to those of joint 2 and 4. This can be interpreted differently, which is reflected in the choice of the coordinate frame. One coordinate frame was designed when the kinematic chain for the youBot was developed, while the other is used for the dynamics chain. It is important for the user to understand in which coordinate frame the software is working. While the situation with 3 different coordinate frames is not ideal, it would be too much of a hassle to recode software to use a different frame. Hence conversion before calling a function in a different frame is easier.

B.1 YouBot coordinate frame

In the youBot coordinate frame, all joints except joint 3 move in positive direction. At calibration, all joints are moved to their physical limits which is then set as the zero position. So for all joints except joint 3 this is their minimum position while for joint 3 it is the maximum position. From this follows that in the youBot coordinate frame joint 3 only has negative positions while the other joints will always be positive.

B.2 Kinematic coordinate frame

The standard youBot coordinate frame switches the direction of rotation around the z axis. This is corrected in this coordinate frame which follows the standard notation for directions found in the literature. It assumes the joints zero position to be when the arm is stretched out in a straight up position.

B.3 Dynamic coordinate frame

The dynamic coordinate frame is used to calculate the torques. It assumes that all joints start at their minimum position and then move in positive direction

CF	Joint	Minimum	Straight up	Maximum
YouBot	1	0	2.9496	5.8992
	2	0	1.1345	2.7053
	3	0	-2.5482	-5.1836
	4	0	1.7890	3.5779
	5	0	2.9234	5.8469
Kinematic	1	2.9496	0	-2.9496
	2	-1.1345	0	1.5708
	3	2.5482	0	-2.6354
	4	-1.7890	0	1.7890
	5	2.9234	0	-2.9234
Dynamic	1	-2.9496	0	2.9496
	2	-1.1345	0	1.5708
	3	-2.5482	0	2.6354
	4	-1.7890	0	1.7890
	5	-2.9234	0	2.9234

Table B.1: Joint Positions in three different coordinate system for three positions.

to their maximum position. Again the zero position is assumed in the straight up position. In addition to the positions, also the velocity and acceleration for joint 3 have to be negated to switch to the other frames.

Bibliography

[1] Rainer Bischoff, Ulrich Huggenberger, and Erwin Prassler. KUKA youBot – a mobile manipulator for research and education. *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, pages 1 – 4, 2011.

[2] Sharat Chandran. Introduction to kd-trees. `http://www.cs.umd.edu/class/spring2002/cmsc420-0401/pbasic.pdf`.

[3] Timothy Jenkel, Richard Kelly, and Paul Shepanski. Mobile manipulation for the kuka youbot platform. https://www.wpi.edu/Pubs/E-project/Available/E-project-031113-133138/unrestricted/Mobile_Manipulation_for_the_KUKA_youBot_Platform.pdf.

[4] Miguel Llama, Rafael Kelly, and Victor Santiba nez. Stable computed-torque control of robot manipulators via fuzzy self-tuning. *IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics, vol. 30*, pages 143 – 150, 2000.

[5] Paul Malmsten. Object discovery with a microsoft kinect. http://www.wpi.edu/Pubs/E-project/Available/E-project-121512-232610/unrestricted/MalmstenRAILMQP.pdf.

[6] Richard Murray, Zexiang Li, and Shankar Sastry. *A Mathematical Introduction to Robotic Manipulation*. CRC Press, 1994.

[7] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, pages 1 – 4, 2011.

[8] Shashank Sharma, Gerhard K. Kraetzschmar, Christian Scheurer, and Rainer Bischoff. Unified Closed Form Inverse Kinematics for the KUKA youBot. *Proc. of ROBOTIK 2012; 7th German Conference on Robotics*, pages 1 – 6, 2012.

[9] Sigurd Skogestad and Ian Postlethwaite. *Multivariable Feedback Control - Analysis and design*. John Wiley & Sons, Ltd, 2005.