



Norwegian University of  
Science and Technology

# AI Planning and Low-Level Control for a Robotic Manipulator

**Øyvind Harding Gulbrandsen**

Master of Science in Cybernetics and Robotics

Submission date: June 2018

Supervisor:           Anastasios Lekkas, ITK

Norwegian University of Science and Technology  
Department of Engineering Cybernetics



---

# Summary

This thesis aims to explore the fusion of high-level AI planning and low-level robotic control. Although the field of AI planning can be said to have sprung out from robotics, researchers have recently expressed concern that the two fields have moved in separate directions even though there is great potential for development in their intersection. This has raised a number of open questions about how to best merge AI planning techniques with robotic control. This project explores some of these questions through the process of building an autonomous robot system.

Three modules formed the building blocks of the system: The control module, the perception module and the planning module. The control module was made by setting up a custom-built robot manipulator and enhancing it with inverse kinematics control of its movements. The perception module made use of a Kinect sensor to take RGB and depth images of the environment and analyzed the images using deep learning in order to detect objects. The perception module also determined the location of the detected objects. The planning module consisted of a STRIPS planner that automatically designed series of actions for the robot to perform in order to bring the environment to a desired state.

The modules were merged into a robot system that was able to operate in a real-world environment. The data about the objects and their location in the environment was fed from the perception module to the planning module in order to create a model of the environment and to initialize the planning algorithm. The continuous execution monitoring performed by the cooperation between the perception and planning modules was shown to be able to handle a number of unexpected situations that happen in the real world. The robot was unavailable during the final tests, but the modular system was easily adjusted for this in order to work for simulated robot actions done by hand. This flexibility of the system would also allow it to be used for any other robotic manipulator by changing the robot measurement specifications.

The goal of creating a system merging AI planning with robotic control is considered achieved. A series of interesting ideas for expanding the system are left open for future research, and it is hoped that these ideas will be explored in the years to come by students and researchers, using the robotic system presented in this thesis as a foundation for their work.

---

---

# Sammendrag

Denne avhandlingen er ment å utforske sammensmeltingen av høynivå AI-planlegging og lavnivå robotstyring. Til tross for at AI-planlegging er et felt som kan sies å ha sitt opphav i robotikk har forskere nylig påpekt at feltene har beveget seg i ulike retninger selv om det er stort potensiale for utvikling nettopp i dette krysningspunktet. Det er fremdeles åpne spørsmål om hvordan denne blandingen mellom AI-planlegging og robotikk kan gjennomføres best mulig. Dette prosjektet utforsker noen av disse spørsmålene gjennom å bygge et automatisk robotsystem.

Tre moduler dannet grunnlaget for robotsystemet: Kontrollmodulen, persepsjonsmodulen og planleggingsmodulen. Kontrollmodulen ble laget ved å sette opp en spesiallaget robotmanipulator som ble utstyrt med et styringssystem basert på invers kinematikk. Persepsjonsmodulen brukte en Kinect-sensor for å ta RGB- og dybdebilder av omgivelsene, og ved bruk av dyp læring ble objektene i bildene detektert. Videre beregnet persepsjonsmodulen objektene sine posisjoner i rommet. Planleggingsmodulen besto av en STRIPS-planlegger som automatisk konstruerte planer bestående av en rekke instruksjoner for roboten, slik at ved å følge instruksjonene ville roboten forandre sine omgivelser for å oppnå et mål.

De tre modulene ble integrert i et felles system som opererte i et virkelig miljø. Data fra persepsjonsmodulen om objektene i området og deres lokasjon ble gitt til planleggingsmodulen for å skape en modell av miljøet og for å initialisere planleggingsalgoritmen. Den kontinuerlige overvåkingen av utførelse gjort av persepsjonsmodulen og planleggingsmodulen viste seg å fungere godt og håndterte en rekke uforventede situasjoner som kan oppstå under arbeid i den virkelige verden. Roboten ikke var tilgjengelig under den siste testperioden, men det modulære systemet ble enkelt forandret til å virke med simulerte robotbevegelser gjort av mennesker i stedet. Denne systemfleksibiliteten gjør det også mulig å bruke en annen robotmanipulator ved å endre på de implementerte robotspesifikasjonene.

Målet om å lage et system som kombinerer AI-planlegging med robotstyring er ansett for å være oppnådd. Rekken med interessante ideer for videreutvikling av systemet som blir presentert i denne avhandlingen blir forhåpentligvis utforsket i de neste årene av studenter og forskere som ønsker å bygge videre på det utviklede systemet.

---

---

# Preface

This master's thesis covers my master's project at the Cybernetics and Robotics programme at the Norwegian University of Science and Technology. The project was carried out during the spring semester of 2018. The supervisor of my thesis was Anastasios Lekkas, associate professor at NTNU. I want to thank Lekkas for giving me a lot of freedom in the project, but still providing interesting discussions and keeping me updated with state-of-the-art research.

The university and my supervisor provided me tools necessary to carry out the research in the thesis. The following is a list of all information, software and apparatus used as a basis for the research as well as the help I received during the project:

- A Dell Optiplex 9010 computer was at my disposal at NTNU, and the university also provided me a large work desk and storage space for my apparatus.
- Python 3 was used for all software implementation, with the exception of changes that were made to the C code of the Libfreenect toolbox.
- My supervisor provided me with a custom built robotic manipulator with actuators from ROBOTIS. He was already in possession of the robot before I began working on the project, and I planned the project based on the possibilities of what could be done with the robot. Along with the robot itself, my supervisor also handed me the essential tools needed for the robot to communicate with a computer: A power cord, signal wires and the USB2Dynamixel communication device. No software was included.
- Staff engineers at the cybernetics department at my university helped me by creating a heavy, circular metal plate that would act as a platform for the robot to stand on. This prevented the robot from tipping over when extending the robot arm.
- A software toolbox named IKPy that contained functionality for calculation of the inverse kinematics of robot models was used to determine the joint positions of the robotic manipulator.
- A Kinect sensor was borrowed from the university. No software was included.

- 
- The software toolbox Libfreenect was used for reading the sensors of the Kinect. My classmate Simon Blindheim provided me instructions on how to install the toolbox and connect it to the Kinect.
  - The OpenCV software library was used for performing simple camera vision tasks of the images taken by the Kinect.
  - YOLOv3 is a software system for real-time object detection in images, and it was used in this project for analyzing images taken by the Kinect.
  - The Pyntcloud software library was used for visualizing point clouds created by RGB images and depth images taken by the Kinect.
  - The Pyperplan software library with its STRIPS planner implementation was used for automatic plan generation.

A detailed description of all hardware and software used in the project is given in Chapter 3.

Nearing the end of the project, but before some of the testing, several of the actuators in the robot were unfortunately severely damaged, rendering the robot unusable for the remainder of the project. The details are described in Section 5.1.1. In the time period after this, with help from my supervisor and the technical staff at the university, the new actuators were ordered, but the delivery was canceled several weeks later due to supply shortage. It was also not possible to utilize similar robotic manipulators at the university. The event affected the system tests as described in Chapter 5.

The task of merging AI planning with robotic control was suggested by my supervisor Lekkas to be the objective of this thesis. As a starting point, Lekkas provided suggestions on how to study relevant theory in the fields of AI planning and robotics to form a theoretical foundation for the project. He recommended me to follow a five-week online course by the University of Edinburgh on AI planning [47], and he also provided a list of relevant research on the intersection of AI planning and robotics.

The project involved setup of the different tool in order to create a complete robot system, and although the objective of the thesis was set, the choice and implementation of software was largely an open task. All of the previously mentioned software toolboxes were integrated into the system by me. As such, the project was designed to be solved individually. The guidance sessions with Lekkas were opportunities to present and discuss the draft of the thesis as well as ideas on how to further progress in the project.



# Table of Contents

<b>Summary</b>	<b>i</b>
<b>Sammendrag</b>	<b>iii</b>
<b>Preface</b>	<b>v</b>
<b>Table of Contents</b>	<b>vii</b>
<b>List of Figures</b>	<b>xi</b>
<b>List of Tables</b>	<b>xiii</b>
<b>Abbreviations</b>	<b>xvii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Description . . . . .	1
1.2 Background and Motivation . . . . .	1
1.3 Objective and Goals . . . . .	2
1.4 Contributions . . . . .	3

---

1.5	Outline . . . . .	4
<b>2</b>	<b>Theory</b>	<b>7</b>
2.1	Robotic Control . . . . .	7
2.1.1	Transformation Matrices . . . . .	8
2.1.2	Inverse Kinematics . . . . .	9
2.1.3	Optimization . . . . .	10
2.2	Camera Calibration . . . . .	13
2.2.1	Intrinsic Camera Parameters . . . . .	13
2.2.2	Distortion Coefficients . . . . .	15
2.3	Artificial Neural Networks . . . . .	16
2.3.1	Forward Pass . . . . .	17
2.3.2	Backward Pass . . . . .	20
2.3.3	Deep Neural Networks . . . . .	21
2.3.4	Convolutional Neural Networks . . . . .	21
2.3.5	Deep Residual Networks . . . . .	24
2.4	Automated Planning . . . . .	25
2.4.1	Defining the Planning Problem . . . . .	25
2.4.2	Planning Languages . . . . .	26
2.4.3	The STRIPS Planning Language . . . . .	27
2.4.4	Search Algorithms . . . . .	28
2.4.5	Heuristics . . . . .	30
2.4.6	Planning in a Real-World Environment . . . . .	32
<b>3</b>	<b>Tools and Libraries</b>	<b>35</b>
3.1	Robotic Manipulator . . . . .	35
3.1.1	Dynamixel Actuators . . . . .	36
3.1.2	Programming Interface . . . . .	38

---

3.2	IKPy and URDF . . . . .	38
3.3	Kinect . . . . .	38
3.4	Libfreenect and Cython . . . . .	39
3.5	OpenCV . . . . .	40
3.6	YOLOv3 . . . . .	40
3.6.1	Darknet-53 . . . . .	41
3.6.2	Architecture . . . . .	41
3.6.3	Python Interface . . . . .	43
3.7	Pyntcloud . . . . .	43
3.8	Pyperplan . . . . .	44
<b>4</b>	<b>Design and Implementation</b>	<b>45</b>
4.1	System Design . . . . .	45
4.2	The Control Module . . . . .	46
4.2.1	Set-up of the Manipulator . . . . .	47
4.2.2	Modelling the Robot . . . . .	50
4.2.3	Inverse Kinematics . . . . .	51
4.2.4	Robotic Control . . . . .	53
4.3	The Perception Module . . . . .	56
4.3.1	Choice of Camera Hardware . . . . .	56
4.3.2	The Kinect Sensors . . . . .	57
4.3.3	Object Detection . . . . .	59
4.3.4	Calibration of the Kinect . . . . .	62
4.3.5	3D Point Localization . . . . .	63
4.4	The Planning Module . . . . .	66
4.4.1	Manual Domain and Goal Specification . . . . .	66
4.4.2	Automatic State Estimation . . . . .	66
4.4.3	Lookahead Planning . . . . .	67

---

<b>5</b>	<b>Testing</b>	<b>69</b>
5.1	Module Tests . . . . .	69
5.1.1	Robot Accuracy and Operational Challenges . . . . .	69
5.1.2	3D Localization Accuracy . . . . .	70
5.1.3	Planning Tests . . . . .	72
5.2	Case Study . . . . .	77
5.2.1	Problem Specification . . . . .	77
5.2.2	Solution . . . . .	78
5.2.3	Handling Complex Situations . . . . .	82
5.2.4	Limitations . . . . .	83
<b>6</b>	<b>Discussion</b>	<b>85</b>
6.1	Goals and Test Results . . . . .	85
6.2	Design . . . . .	86
6.3	Future Work . . . . .	87
6.3.1	Control Module . . . . .	87
6.3.2	Perception Module . . . . .	87
6.3.3	Planning Module . . . . .	88
6.3.4	Additional Functionality . . . . .	89
<b>7</b>	<b>Conclusion</b>	<b>91</b>
	<b>Bibliography</b>	<b>93</b>

# List of Figures

2.1	Two coordinate systems A and B . . . . .	8
2.2	Three joints on a robotic manipulator . . . . .	10
2.3	Pinhole camera model . . . . .	14
2.4	ANN . . . . .	16
2.5	Perceptron . . . . .	18
2.6	Human neuron . . . . .	18
2.7	Convolutional layer . . . . .	22
2.8	Average pooling . . . . .	23
2.9	Residual block . . . . .	24
2.10	PDDL domain and problem specification example . . . . .	29
2.11	Planning graph . . . . .	32
3.1	Side view of the robotic manipulator . . . . .	36
3.2	Manipulator grippers . . . . .	37
3.3	The Kinect . . . . .	39
3.4	YOLOv3 architecture . . . . .	42

---

4.1	System overview . . . . .	46
4.2	PID controller . . . . .	48
4.3	Manipulator model . . . . .	50
4.4	Manipulator model . . . . .	52
4.5	Three coordinates systems used for robotic control . . . . .	53
4.6	Modeled and physical manipulator position . . . . .	55
4.7	Kinect color and depth images . . . . .	58
4.8	Kinect coordinate system . . . . .	59
4.9	Detection of several objects with YOLOv3 . . . . .	60
4.10	Stability of YOLOv3 . . . . .	62
4.11	RGB image mapped to depth plane . . . . .	63
4.12	Point cloud visualization . . . . .	64
4.13	Separate objects on the depth image plane . . . . .	65
5.1	3D localization using the mean . . . . .	70
5.2	3D localization using the median . . . . .	71
5.3	Simple planning problem . . . . .	73
5.4	Difficult planning problem . . . . .	74
5.5	Search algorithm performance on a blocks world-like problem . . . . .	76
5.6	The setup of the fruit stacking problem . . . . .	77
5.7	Object detection in the fruit stacking problem . . . . .	79
5.8	The movement of the robot . . . . .	80
5.9	Sequence diagram of the control flow . . . . .	81

# List of Tables

2.1	Common ANN activation functions . . . . .	19
4.1	Stability of YOLOv3 . . . . .	61
5.1	Performance on a simple planning problem . . . . .	73
5.2	Performance on a difficult planning problem . . . . .	75
5.3	System time analysis . . . . .	82

---



# List of Algorithms

2.1	BFGS . . . . .	12
2.2	Run-Lookahead . . . . .	33

---

---

# Abbreviations

AI	=	Artificial intelligence
ANN	=	Artificial neural network
DNN	=	Deep neural network
CNN	=	Convolutional neural network
CPP	=	Classical planning problem
STRIPS	=	Stanford Research Institute Problem Solver
PDDL	=	Planning Domain Definition Language
HTN	=	Hierarchical task networks
BFS	=	Breadth-first search
DFS	=	Depth-first search
GBF	=	Greedy best-first
RL	=	Reinforcement learning

---

# Chapter 1

## Introduction

### 1.1 Problem Description

Robots are fast, accurate and reliable, making them perfect for performing routine tasks. However, if a robot needs to perform complex tasks in a constantly changing environment, hard-coding its behavior may prove difficult. Utilizing Artificial Intelligence (AI) opens up for great possibilities: The robot can adapt to the environment in order to optimize its behavior in real time.

How to **fuse high-level AI and low-level robotic control** in a meaningful way is the problem that will be explored in this master's thesis.

### 1.2 Background and Motivation

High-level AI can enrich the field of robotics in a number of areas: Situation reasoning, perception, human interaction, robot interaction and self-improvement through learning are just a few of the possible improvements to robotics made possible by artificial intelligence techniques such as AI planning, artificial neural networks and reinforcement learning [19].

The field of AI planning is originally rooted in robotics: Considered one of the

first AI planning languages, STRIPS was designed to automate the robot Shakey at the Stanford Research Institute in the early 1970s [10]. Shakey used a camera to observe its surroundings, and its STRIPS planner allowed it to automatically generate plans consisting of sequential actions in order to fulfill orders from humans [48]. The field of AI planning developed in the following decades, greatly improving the expressivity and speed of planning algorithms thanks to the development of modern search algorithms and heuristics. However, researchers have recently pointed out that theoretical areas of AI planning are over-explored, whereas the practical merging of AI planning with robotics needs to be put in focus [35][12][11][29][19]. Rajan and Saffiotti [35] mention three reasons as to why this fusion is ready to be addressed now: Firstly, advances in hardware and sensors allow embedded sensors to run advanced programs. Secondly, there are large amounts of data sets available on the internet that can be used to train machine learning methods. Finally, the fields of robotics and AI have matured separately, and so researchers should be able to evaluate which of the developed AI techniques might be useful in robotics.

Ghallab et al. [12] present a large number of open problems connected to the fusion of AI planning and robotics: How to model the environment, how to perform monitoring with prediction and observation discrepancy detection and plan recovery techniques, and how to integrate a full system with real-time constraints, are some of the questions that they want to see further explored through research. This project explores these questions through the construction of an automatic robot manipulator system. It uses recently developed deep learning methods for object detection as a basis for observation, as was suggested by Puja et al. [33]

It is hoped that this thesis will help contribute towards the fusion of AI planning and robotic control: An automatic robot system that can be used in future research by other students and researchers has been created. The same robot will be used by students who will be writing their master's thesis at NTNU next year, and hopefully it will be used by students and researchers for even more years to come. It is hoped that the contributions made by this thesis will serve as a foundation for those studies, and that this thesis report will serve as a valuable resource for the theoretical background of the methods used as well as a pointer on interesting directions in future work.

### 1.3 Objective and Goals

The thesis' objective is to build a novel system that fuses AI and robotic control. Specifically, the system will combine control with high-level AI planning. However, com-

pleting this system is but one of the larger goals of this thesis. The goals of the project can be summed up as follows:

- A study and understanding of fundamental theory and current research on blended AI planning and acting.
- Setup and implementation of relevant hardware and software tools.
- Completion of a full working system that fuses AI and robotic control.

The objective was broken down into several smaller tasks, creating a project plan:

1. Study current research on integration of AI planning and robotic control and which solutions exist. This includes revisiting my specialization report from the fall in 2017 [15], which was a study on reinforcement learning and deep learning in the context of robotics.
2. Setup of the robot.
3. Choice and implementation of robot control methods and software in order to move the robot arm as desired.
4. Choice and implementation of sensor and software so that the robot can react to the environment.
5. Study fundamental AI planning through an online course [47].
6. Choice of planning software and its integration into the robot system.
7. Completion of the automatic robot control system and testing it in a real-world setting.

## 1.4 Contributions

The following is a description of the specific contributions made by this project:

- A robot manipulator was set up using an SDK developed by ROBOTIS. A software toolbox solving the inverse kinematics problem was then integrated in order to control the robotic actuators. This is described in Section 4.2.

- A Kinect sensor was set up, and the YOLOv3 object detection software was applied to images of the Kinect. A system for estimating the position of every detected object in these images was implemented based on the libfreenect toolbox. 3D visualizations of the images taken by the Kinect were created with the help of the pyntcloud SDK. These contributions are presented in Section 4.3.
- Using the pyperplan toolbox, planning domains and problems relevant for a robotic manipulator were designed for the AI planner. A system was implemented for automatically translating information about objects and their position into a problem description that was readable for the AI planner. This is described in Section 4.4.
- The implemented tools were combined into a full robotic system. The system was able to perform execution monitoring by continuously evaluating and storing an updated model of the environment, keeping track of the changes in the world state after having performed actions with the robot, and finally replanning if unexpected events had occurred. The finalized system is flexible and can easily be changed in order to work with any robotic manipulator.

## 1.5 Outline

The thesis begins with a presentation on the fundamental theory behind methods and tools used in the project in Chapter 2. Theory on inverse kinematics and optimization algorithms used for robotic control is explained first, in Section 2.1. Thereafter, a short explanation on camera calibration is given in Section 2.2, before theory on artificial neural networks is presented in Section 2.3. Finally, an introduction to automated planning is given in Section 2.4.

In Chapter 3, an overview of the tools and libraries used for the implementation in the project is presented.

Implementation of the project is presented in Chapter 4. The chapter begins with a presentation of the system design in Section 4.1, which explains how the system was divided into an actor, a perception and a planning module. The implementation of the actor module is described in Section 4.2. Thereafter, Section 4.3 presents the perception module, before the planning module is described in Section 4.4.

A presentation of system tests and their results is presented in Chapter 5. In Section 5.1, the individual modules of the systems were tested, whereas Section 5.2 presents a case study of the full system.



Chapter 6 discusses on the design, main results and possible improvements that can be implemented in the future.

Finally, Chapter 7 concludes the thesis and presents a concise list on suggestions for future work.



# Chapter 2

## Theory

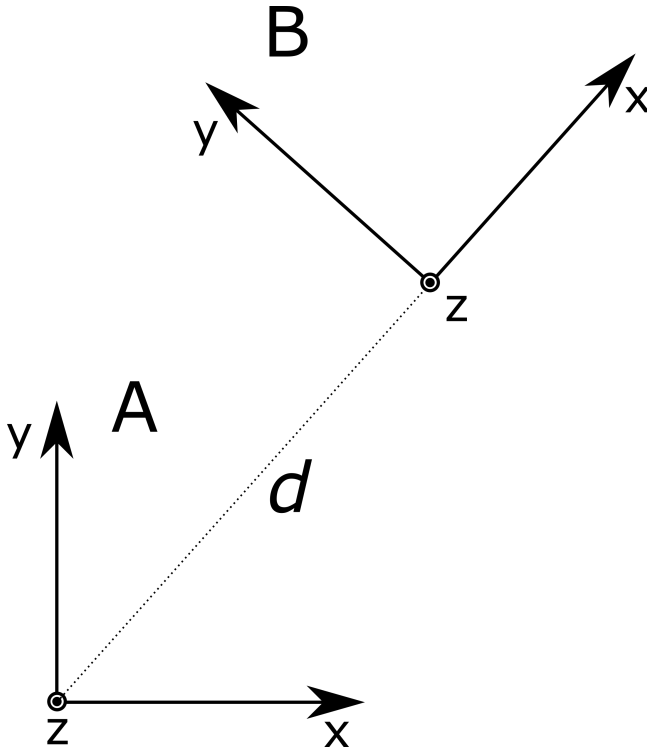
This chapter presents relevant theory that lays the foundation for the implementation of the robotic system that was designed in this thesis. The theory is divided into four parts:

- Section 2.1 covers the theory needed to control the movement of the robotic manipulator. Control of the robotic manipulator is the actor part of the system design.
- Section 2.2 and Section 2.3 cover theory for the perception module of the system. These chapters present theory on camera geometry and calibration, as well as object detection.
- Section 2.4 present theory behind planning algorithms. This is the foundation of the planning module in the system design.

### 2.1 Robotic Control

A robotic manipulator consists of a series of joints in which pairs of joints are connected by a rigid structure. It is assumed in this thesis that all of the robotic joints are **revolute**, which means that their only available movement is revolution about one axis.

### 2.1.1 Transformation Matrices



**Figure 2.1:** Coordinate systems A and B. Aligning the two coordinate systems requires translation and rotation. The translation,  $d$ , is the distance between the origins of the two systems.

Two coordinate systems,  $A$  and  $B$ , are shown in Fig. 2.1. The origins of the two coordinate systems do not overlap, and so there is a **translation** between the coordinate systems. The translation can be denoted by

$$d = \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} \quad (2.1)$$

where  $d$  is as seen in Fig. 2.1, and  $d_x$ ,  $d_y$  and  $d_z$  are the components of  $d$  along the  $x$ ,  $y$  and  $z$  axes.

However, the axes of the coordinate systems point in different directions. A **rotation** is also needed in order to describe the coordinate system difference:

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \quad (2.2)$$

where the elements of  $R$  can be found by decomposing the rotation into successive rotations about the principal coordinate axes. If  $R$  is decomposed into three rotations, firstly about the fixed  $x$  axis, then about the fixed  $y$  axis, and finally the fixed  $z$  axis,  $R$  is given by:

$$R = R_z R_y R_x \quad (2.3)$$

The translation and rotation of (2.1) and (2.2) can be combined into a single matrix. Spong et al. [46] refer to this as the homogeneous transformation matrix,  $H$ :

$$H = \begin{bmatrix} R & d \\ 0 & 1 \end{bmatrix} \quad (2.4)$$

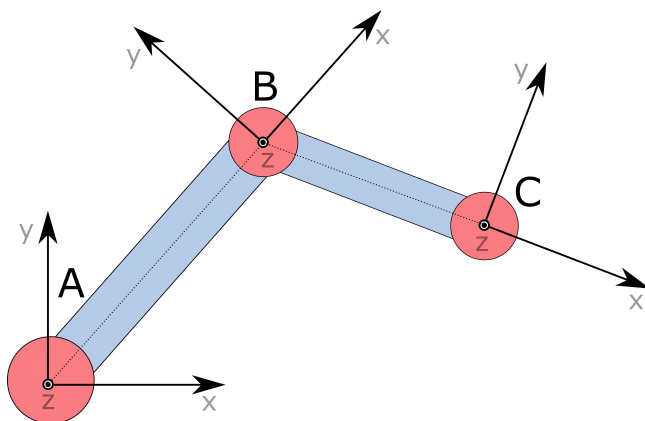
In Fig. 2.1,  $R$  and  $d$  in  $H$  describe the rotation and translation of coordinate system  $B$  given in the frame of coordinate system  $A$ .

### 2.1.2 Inverse Kinematics

The power of (2.4) becomes apparent when used in a chain of transformations. In Fig. 2.2, three joints of a typical robotic manipulator is visualized. The transformation matrix for each joint is defined as follows: Let the transformation from joint  $A$  to joint  $B$  be denoted by  $H_B^A$ , and the transformation from joint  $B$  to joint  $C$  be denoted by  $H_C^B$ . Then the coordinate system at joint  $C$  expressed in coordinate system  $A$  is simply given by  $H_B^A H_C^B$ .

The coordinate system of the first joint can be labeled the **base coordinate system**. When analyzing a robotic manipulator, the coordinate system of the gripper, joint  $n$ , expressed in the base coordinate system at joint 0, is given by:

$$H_n^0 = H_1^0 H_2^1 \dots H_n^{n-1} \quad (2.5)$$



**Figure 2.2:** Three joints of a robotic manipulator are visualized. Each joint is in the origin of its own coordinate system. It can be seen that the length of the translation between two joints is constant, although the decomposition varies according to the angle of the joints. It should also be noted that rotation of one joint rotates all of the coordinate systems further out on the robotic arm.

The transformation matrices from one joint to the next fully describes its angle, and thus all transformation matrices describe the full position of the robot. Eq. (2.5) calculates the forward kinematics of the system: The orientation and location of the grippers is found in the base coordinate system. Inverse kinematics is the opposite task: Given a goal state, a desired orientation and location of the grippers in the base coordinate system, the inverse kinematics problem is concerned with finding a set of joint angles that satisfies the goal state.

### 2.1.3 Optimization

Following the notation of Nocedal and Wright [31], an optimization problem is defined as follows:

$$\begin{aligned}
 & \min_{x \in \mathbb{R}^3} && f(x) \\
 & \text{subject to} && c_i(x) = 0, \quad i \in \mathcal{E} \\
 & && c_i(x) \geq 0, \quad i \in \mathcal{I}
 \end{aligned} \tag{2.6}$$

where  $x$  is a set of unknown variables,  $f(x)$  is the **objective function** that is to be minimized, and  $c_i$  are constraints on the variables in  $x$ . Optimization algorithms are concerned with finding the point  $x$  that minimizes  $f(x)$ . As the objective function might be highly nonlinear, which makes it difficult to find a global minimal point, most optimization algorithms are concerned with finding a locally optimal solution to the optimization problem.

A simplified version of the optimization problem of Eq. (2.6) is the unconstrained problem. This problem has no constraints  $c$  on the variables. One approach to find a locally optimal solution to this problem is to iteratively find good directions to move in; directions in which the objective function decreases. This is called a **line search**. Moving along the vector  $p$  away from the point  $x_k$  changes the objective function from  $f(x_k)$  to  $f(x_k + p)$ .

The following function  $m_k(p)$  can be defined around a point  $x_k$ :

$$m_k(p) = f(x_k) + p^T \nabla f(x_k) + \frac{1}{2} p^T \nabla^2 f(x_k) p \quad (2.7)$$

$m_k(p)$  is actually the approximation of the objective function  $f$  around the point  $x_k$  found by the second-order Taylor series approximation [31]:  $f(x_k + p) \approx m_k(p)$ . More specifically,  $m_k$  is a quadratic function estimate of the objective function around the point  $x_k$ . If  $\nabla^2 f(x_k)$  is positive, the minimal point of  $m_k(p)$  is found when the derivative of  $m_k(p)$  is 0. Derivation of Eq. (2.7) and solving for  $p$  gives

$$p_k = -(\nabla^2 f(x_k))^{-1} \nabla f(x_k) \quad (2.8)$$

where  $p_k$  is found to minimize the objective function estimation:  $m_k(p_k) \leq m_k(p), \forall p$ .

## L-BFGS-B

Calculating  $\nabla^2 f(x_k)$  at every step of an optimization algorithm is computationally hard. The Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm instead calculates an approximation:  $B_k \approx \nabla^2 f(x_k)$ . This estimate is found by the following formula:

$$B_{k+1} = B_k - \frac{B_k(x_{k+1} - x_k)(x_{k+1} - x_k)^T B_k}{(x_{k+1} - x_k)^T B_k (x_{k+1} - x_k)} + \frac{(\nabla f(x_{k+1}) - \nabla f(x_k))(\nabla f(x_{k+1}) - \nabla f(x_k))^T}{(\nabla f(x_{k+1}) - \nabla f(x_k))^T (x_{k+1} - x_k)} \quad (2.9)$$

Replacing  $\nabla^2 f(x_k)$  in equation Eq. (2.8) for  $B_k$ , the search direction is found as follows:

$$p_k = -B_k^{-1} \nabla f(x_k) \quad (2.10)$$

The full BFGS algorithm for minimizing the objective function is described in Algorithm 2.1. Choosing  $B_0$  at Line 2 in Algorithm 2.1 might be done by estimating  $\nabla^2 f(x_0)$ . Another choice is setting  $B_0$  to be the identity matrix,  $B_0 = I$ . At Line 6 in Algorithm 2.1, the variable  $\alpha_k$  is a scaling factor of how much  $x$  should move in the direction of  $p_k$ . It can be set to 1, or it can be adjusted as long as the movement satisfies the constraints known as the Wolfe conditions [31].

```
1: procedure BFGS( $f, x_o, \epsilon$ )
2:   Choose  $B_0$ 
3:    $k \leftarrow 0$ 
4:   while  $\|\nabla f\| < \epsilon$  do ▷ Continue until the slope is satisfyingly flat
5:     Compute  $p_k$  from equation Eq. (2.10)
6:      $x_{k+1} \leftarrow x_k + \alpha_k p_k$ 
7:     Compute  $B_{k+1}$  from equation Eq. (2.9)
8:      $k \leftarrow k + 1$ 
9:   end while
10:  return  $x_k$ 
11: end procedure
```

#### Algorithm 2.1: BFGS

L-BFGS (Limited-memory BFGS) is a variation of the BFGS which stores the vectors used to compute  $B^{-1}$  instead of storing  $B^{-1}$  itself. This is done in order to limit the use of memory. L-BFGS-B is a modification of L-BFGS which is able to solve the optimization problem in Eq. (2.6) also when simple lower and upper bounding constraints are added for each individual variable  $x_i$ .

The L-BFGS-B algorithm can be used to solve the inverse kinematics presented in Section 2.1.3, as will be presented in Section 4.2.3.



## 2.2 Camera Calibration

When a picture is taken by a camera, points in 3D space are mapped onto a 2D film inside the camera. This mapping can be described by a combination of

- **The intrinsic camera parameters:** The mapping from 3D space to a 2D plane modeled by a simple pinhole camera.
- **The distortion coefficients:** Distortion due to the camera lens.

Camera calibration is the process of finding these parameters and coefficients. When knowing the distortion coefficients of a camera, its images can be undistorted so that straight lines in the real world appear straight in the images. This is important because distortion in an image prevents us from computing distances correctly within the image. If a camera is used as the perception module of a control system, it is helpful that the images are undistorted first so that distances can be calculated before performing control tasks.

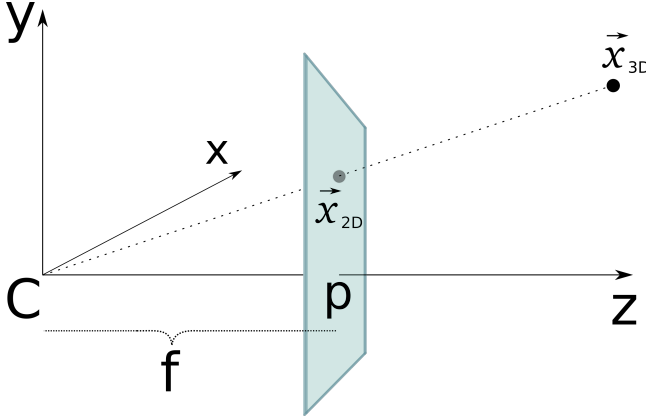
### 2.2.1 Intrinsic Camera Parameters

A pinhole camera model is seen in Fig. 2.3. The image that is stored on the camera is the projection of points onto the image plane. The line passing through the camera center  $C$ , perpendicular to the image plane, is the principal axis  $z$ . The axis passes through the image plane at the principal point  $p$  at  $z = f$ , where  $f$  is the focal length of the camera. Assume  $\vec{x}_{3D}$  is a point in 3D space given by

$$\vec{x}_{3D} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (2.11)$$

that we want to project onto the image plane. By geometric inspection, the projection onto the image plane is given by

$$\vec{x}_{2D} = \begin{bmatrix} fx/z \\ fy/z \\ f \end{bmatrix} \quad (2.12)$$



**Figure 2.3:** The pinhole camera model. In reality, the image plane is on the other side of the camera center  $C$ . This means that the plane should be at  $z = -f$ . However, that inverts the image. By simply rotating the inverted image  $180^\circ$ , this is corrected, and that produces the same image as would be projected onto the image plane in this model. Therefore, this model is just as valid and more simple to visualize.

In homogeneous coordinates, the coordinates of the projected point onto the image plane can be expressed by

$$\vec{x}_{2D,HC} = \begin{bmatrix} fx \\ fy \\ z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \vec{x}_{3D} \quad (2.13)$$

If  $p$  is not at the center of the coordinate system of the image plane, offsets  $p_x$  and  $p_y$  need to be added to  $\vec{x}_{2D}$ . Moreover, in practice two different focal lengths are often used. This is because the image pixels often are rectangular instead of square [7]. Eq. (2.13) can therefore be modified to take these offsets into account:

$$\vec{x}_{2D,HC} = \begin{bmatrix} f_x x + z p_x \\ f_y y + z p_y \\ z \end{bmatrix} = \begin{bmatrix} f_x & 0 & p_x \\ 0 & f_y & p_y \\ 0 & 0 & 1 \end{bmatrix} \vec{x}_{3D} \quad (2.14)$$

Equation (2.14) describes the intrinsic camera parameters, ie. the projection of points in 3D space onto a 2D plane at focal length  $f$  in front of the camera center  $c$ .

### 2.2.2 Distortion Coefficients

The lens of a camera introduces distortion to the ideal model in Eq. (2.14) due to the bending of the light while passing through the lens. Radial and tangential distortion are two effects that our model can compensate for in order to improve. Radial distortion is distortion that changes depending on the distance from the center of the image. Using 3 coefficients to model the distortion, the corrected image coordinates are given by

$$\begin{bmatrix} x'_{rad,corr} \\ y'_{rad,corr} \end{bmatrix} = \begin{bmatrix} x'(1 + k_1r^2 + k_2r^4 + k_3r^6) \\ y'(1 + k_1r^2 + k_2r^4 + k_3r^6) \end{bmatrix} \quad (2.15)$$

where  $r = (x')^2 + (y')^2$ , and  $x'$  and  $y'$  are the coordinates in the image plane given by Eq. (2.14).

Tangential distortion occurs when the image plane and lens are not perfectly parallel. The distortion is given by

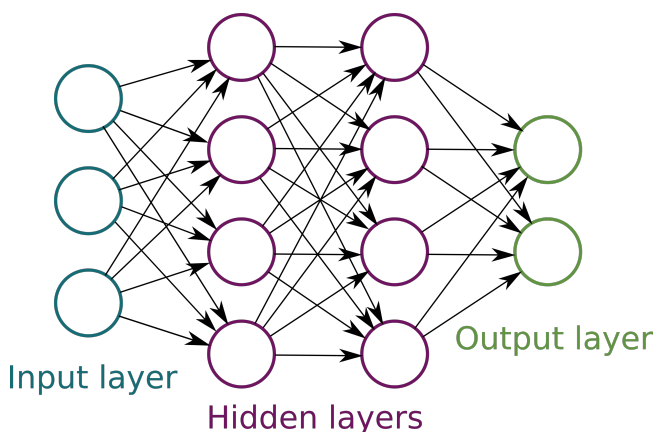
$$\begin{bmatrix} x'_{tan,corr} \\ y'_{tan,corr} \end{bmatrix} = \begin{bmatrix} x' + 2p_1x'y' + p_2(r^2 + 2(x')^2) \\ y' + p_1(r^2 + 2(y')^2) + 2p_2x'y' \end{bmatrix} \quad (2.16)$$

The distortion coefficients of Eq. (2.15) and Eq. (2.16) are as follows:

$$d = \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ p_1 \\ p_2 \end{bmatrix} \quad (2.17)$$

## 2.3 Artificial Neural Networks

ANNs are systems that are designed to simulate the neural system of human brains [1]. An ANN is a special instance of a graph, in which nodes are connected by directed edges. The nodes are divided into **layers**. Edges usually connect nodes in one layer to nodes in the next layer, starting by connecting the **input layer** to the first **hidden layer**, and ending up by connecting nodes in the last hidden layer to nodes in the **output layer**. In a fully connected ANN, every node in one layer is connected to each node in the next. If there are no loops in the network, it is called a feed-forward ANN. An illustration of a fully connected feed-forward ANN with two hidden layers is shown in Fig. 2.4 The nodes are simulating neurons in the human brain: Nodes are transferring information through the edges, similar to how brain neurons fire electrical signals to communicate.



**Figure 2.4:** An ANN with an input layer of three ,  $I_1$ ,  $I_2$  and  $I_3$ , two hidden layers with four nodes in each layer, and an output layer of two nodes,  $O_1$  and  $O_2$ . The ANN can be viewed as a function that takes values on the input nodes to produce a value on the output nodes:  $O_1 = f_1(I_1, I_2, I_3)$  and  $O_2 = f_2(I_1, I_2, I_3)$

By providing data to an ANN, it will return a prediction. It can be treated as a function, where the user provides the input and the ANN gives an output. To train an ANN, we have to use labeled data, which is a set of input and output data. The output data is the correct results corresponding to the input data. During training, an ANN will adjust its own parameters, its weights and biases, until it can provide results that are satisfyingly close to the correct results. A common algorithm for training ANNs is

**backpropagation**, which consists of alternating forward and backward passes through the network.

### 2.3.1 Forward Pass

When viewing the ANN as a function, we specify the input by putting values on the nodes in the input layer. The ANN will then perform a forward pass, which propagates through the edges of the ANN. The output of the function is the value of the nodes in the output layer. To see how it happens, it is necessary to look at what happens to a node in layer  $k$  during the propagation from layer  $k - 1$ .

Every edge in the network has a **weight**, which is multiplied by the value of its source node in layer  $k - 1$  to give an input value to the sink node in layer  $k$ . The sink node might be a sink to many edges. In addition, the sink node gets one input value from a **bias node**. The resulting function is known as the perceptron function:

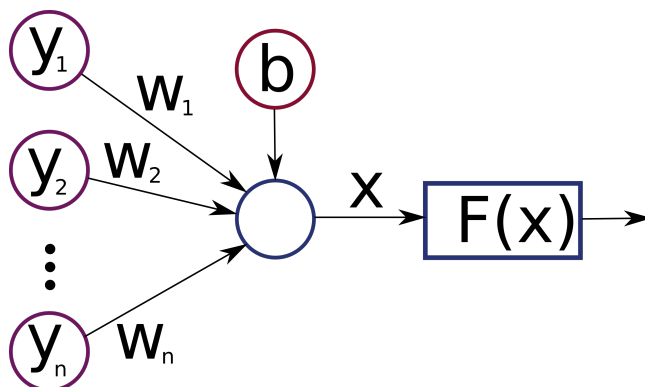
$$x = \sum_{k=1}^n [y_k w_k] + b \quad (2.18)$$

where  $x$  is the input to the node.  $\sum(y_k w_k)$  is the sum of all the values of the nodes in layer  $k$  multiplied by the weights connecting them, and  $b$  is the bias. The node uses the output of Eq. (2.18) as input for its **activation function**, and the result of becomes the output signal from the node the node. A visualization of the perceptron is shown in Fig. 2.5. The similarities to human neurons is striking, as visualized in Fig. 2.6.

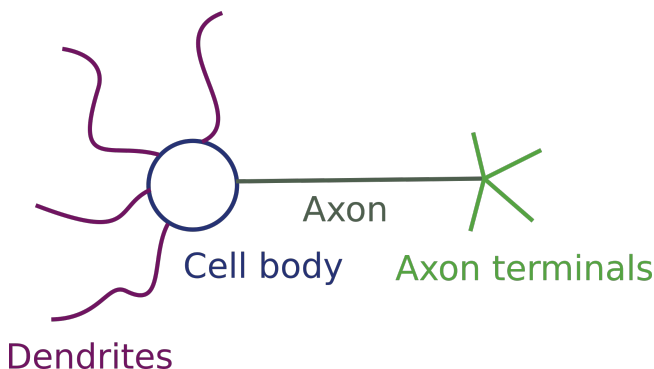
#### Activation Functions

Equation (2.18) might give any real-valued number. We can consider this the input to a node. The node then calculates its activation, how much it fires up, from this input. This calculation is the activation function. The activation function is important for one reason: It introduces non-linearities to the ANN. Without using activation functions, the output could only be a linear function of the input of the network. Moreover, the full ANN could be reduced to a single layered network, because the combination of linear functions could be reduced to a single linear function of the input layer.

A simple activation function is a step function that takes the value 1 when Eq. (2.18) is larger than 0, and 0 otherwise. However, the binary nature of such a function cannot be varied gradually, which is inconvenient when adjusting the ANN weights during a



**Figure 2.5:** The perceptron function in Eq. (2.18) is visualized for a single node in the ANN.  $y_1, y_2, \dots, y_n$  are the output signals of the input nodes. The output signal of the node is found through applying the activation function  $F$  to the output of Eq. (2.18).

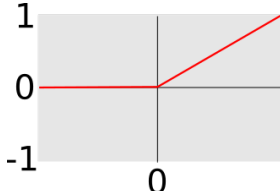
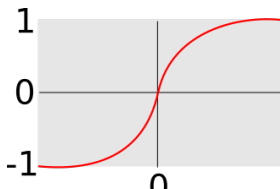



**Figure 2.6:** This is a simplified model of a human neuron. The dendrites receive signals from other neurons and transfer them to the cell body. The cell body might "fire" if stimulated by the input signals from the dendrites. When the neuron fires, it sends a signal to the axon terminals, which distribute the signal to other neurons. This process is very similar to the artificial neuron function showed in Fig. 2.5. This visualization is inspired by Wolfe et al. [49]

backward pass. A linear activation function is also not desired, because its derivative is always constant. This is also a problem during a backward pass.

There are a variety of activation functions with desirable properties for the back-

ward pass: ReLU, tanh, sigmoid and softmax are commonly used activation functions. They are shown in Table 2.1

Common activation functions		
Function	Formula	Figure
ReLU	$F(x) = \max(0, x)$	
Tanh	$F(x) = \tanh(x)$	
Sigmoid	$F(x) = \frac{1}{1 + \exp x}$	
Softmax	$F(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$	

**Table 2.1:** A table of common activation functions and their plots.

The ReLU activation function is given by

$$F(x) = \max(0, x) \quad (2.19)$$

One advantage of the ReLU activation function is that the node does not fire if  $x < 0$  in Eq. (2.19). This makes computation easier because there will be many 0's in the network. For huge ANNs with many nodes this might be very beneficial for computational time.

If the values of the output nodes represent probabilities, the softmax activation function is often used for the output layer nodes. It normalizes the values such that the sum of all output nodes is equal to 1. The calculation for each output node  $i$  by softmax is given by

$$F(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \quad (2.20)$$

where  $j$  is an index running through all of the output nodes.

### 2.3.2 Backward Pass

After performing a forward pass, the output of the ANN might be different from the ideal output. The **error** of the output is a measure of the difference between the output of the ANN and the ideal output. The backward pass calculates the gradient of the error function with respect to every weight and bias in the network. We define the gradient by

$$\begin{aligned} \nabla E_\omega &= \left[ \frac{\partial E}{\partial \omega_1}, \frac{\partial E}{\partial \omega_2}, \dots, \frac{\partial E}{\partial \omega_p} \right] \\ \nabla E_b &= \left[ \frac{\partial E}{\partial b_1}, \frac{\partial E}{\partial b_2}, \dots, \frac{\partial E}{\partial b_q} \right] \end{aligned} \quad (2.21)$$

for  $p$  number of weights and  $q$  number of bias nodes in the ANN. Because the gradient points towards the direction of greatest ascent, each weight needs to move in the opposite direction of the gradient in order to move towards a minimum. Every weight and bias node in the network is therefore updated by

$$\begin{aligned} \omega_i &\leftarrow \omega_i - \alpha \frac{\partial E}{\partial \omega_i} \\ b_i &\leftarrow b_i - \alpha \frac{\partial E}{\partial b_i} \end{aligned} \quad (2.22)$$

The partial derivatives in one layer are dependent on partial derivatives in layers that are closer to the output. Therefore, the calculation is done from the output node and backwards.



The choice of error function depends on the output. If the output is a normal scalar, the error function can be found as a square error:

$$E = \sum_i [(z_i - y_i)^2] \quad (2.23)$$

where  $i$  runs through the output nodes,  $y_i$  is the output of node  $i$ , and  $z_i$  is the ideal output of node  $i$ . If the output represents probabilities, the error of the output can be found by a cross-entropy function:

$$E = \sum_i [z_i \ln y_i + (1 - z_i) \ln(1 - y_i)] \quad (2.24)$$

The reason why Eq. (2.24) is an effective error function, is that the update step in Eq. (2.22) of the weights and bias leading into the node giving the output error in Eq. (2.24) is proportional to the difference between  $z$  and  $y$  when the output values are formed by a sigmoid or softmax activation function [30].

### 2.3.3 Deep Neural Networks

DNNs are ANNs that are deep, ie. networks with more than one hidden layer. The depth of an ANN is the number of hidden layers + the output layer in the network. It has been shown that an ANN with only one hidden layer can approximate any continuous function with as high precision as desired, provided no constraints on the weights or the number of nodes [8].

If an ANN with only one hidden layer can be used to learn any function, why do we need DNNs? Bengio [5] provides two answers: Firstly, deep networks are observed to generalize well. This is important to avoid overfitting of the learned function to the training data. Secondly, a function learned by a deep network might need very many more neurons per layer to be represented by a shallow network. The benefit is therefore also computational.

### 2.3.4 Convolutional Neural Networks

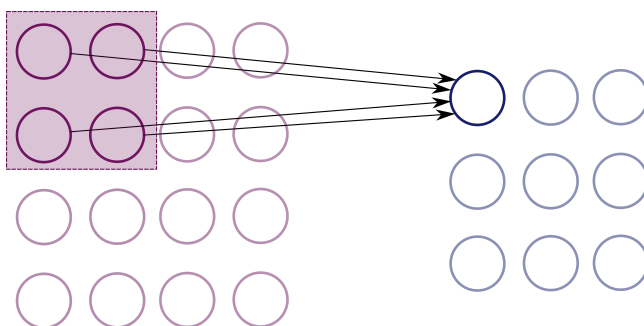
CNNs are feed-forward DNNs that have been shown to perform very well at visual classification tasks [50]. The input to CNN used for visual classification is normally

a matrix containing the intensity of every single pixel in the image. Unlike fully connected DNNs, CNNs uses the spacial information of these pixels, which may explain why they perform so well.

There are many ways to design CNNs, but they need to contain convolutional layers. Often they also contain pooling layers, either in the form of maxpooling layers or averaging layers.

### Convolutional Layer

In a convolutional layer, every node in one layer is not connected to all of the nodes in the next layer. Instead, only a region of nodes connect to each node in the next layer. This region is called a **filter**. A convolutional layer is visualized in Fig. 2.7.



**Figure 2.7:** One convolutional layer is visualized. A filter of size 2x2 is marked by a purple square. All nodes covered by the filter in the first layer contribute to the value of one node in the second layer. The weights of the edges are fixed by the filter. The filter slides across all of the nodes in the first layer. The activation function for the nodes in the second layer is usually the ReLU activation function.

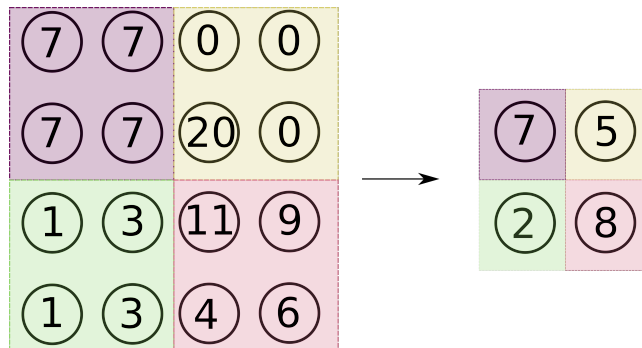
As can be seen in Fig. 2.7, the number of nodes has decreased after the convolutional operation. The bigger the filter, the smaller the second layer would be. In order to keep the second layer the same size as the first layer, zero padding of the first layer can be performed. Zero padding means that the first layer is expanded by adding nodes of value 0. This way, by adding some exact number of zero valued nodes, the second layer can be kept the same size as the first layer before padding.

It is important to note that the weights belong to the filter, and not the individual nodes in the first layer. It is also the weights of the filters that are trained in the

backward pass. For the first convolutional layer, the one acting directly on the image input pixels, the filters have a simple interpretation. The weights of a filter makes the filter represent a shape, and by sliding the filter across the input layer, the filter picks up regions where this shape occur in the image. A convolutional layer typically has several filters, making it possible to detect several shapes. The filters in next convolutional layers connect these simple shapes into more and more complex geometrical shapes. Zeiler and Fergus showed visually how this complexity grows deeper into the convolutional network [50].

### Pooling Layer

A pooling layer is a form of downsampling. The nodes are divided into regions of a fixed size. One type of pooling layer is an average pooling layer, which finds the average value of every region. This is shown in Fig. 2.8. Another usual pooling type is maxpooling, which only keeps the node with the biggest value of every region.



**Figure 2.8:** An average pooling layer is visualized. In this example, filters of size 2x2 are used to divide the nodes into regions.

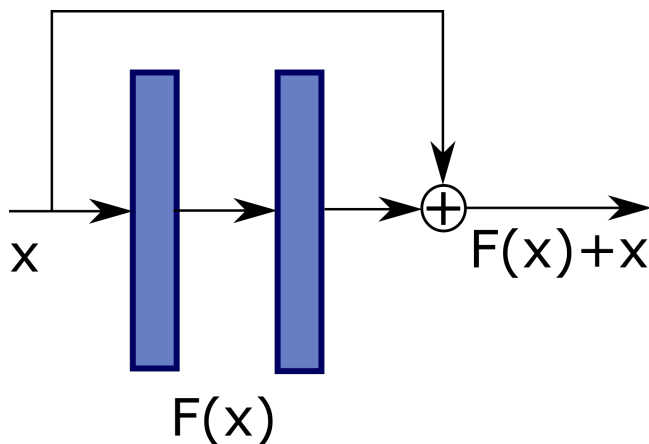
There are several reasons for why pooling is performed [51]. Firstly, reducing the spatial information allows for faster computation time. Secondly, pooling makes the visual classification more space invariant. When certain shapes are found in the image, it does not necessarily matter exactly where in the image the shapes are located. What is more important is how the shapes are located relatively to each other when they build up object together. Pooling keeps the relative location of the shapes intact while making the absolute location of the shapes in the image less important.

Global average pooling is an extreme form of average pooling in which the filter is the same size as the node layer. That means that a global average pooling layer averages all of the nodes into a single node.

### 2.3.5 Deep Residual Networks

As introduced in subsection 2.3.3, deep networks are able to generalize well in order to avoid overfitting to training data. However, as shown by He and Sun [16], adding more and more depth to the network is not always a good idea. At some point the accuracy of a neural network does not improve when adding layers, and might even be reduced. Training deep neural networks is difficult.

Deep residual networks, presented by Microsoft researchers in 2015 [17], introduce a way to circumvent this problem: Early layers are copied and inserted further down the pipeline. This is illustrated in Fig. 2.9. A residual network contains one or more of these feed-forward connections. The layers in between a feed-forward mapping are used to calculate a **residual**. It is hypothesized that the residual  $F(x)$  is easier to calculate than if there was no such mapping. Theoretically, if  $x$  is already optimal, then the optimal residual is:  $F(x) = 0$ . It is easier to train the residual to go towards 0 than it would be to train a network with no mapping so that the same part of the network would output  $x$ .



**Figure 2.9:** A residual block. The layer  $x$  is mapped to a point later down in the pipeline. The two layers in between the feed-forward are trained to optimize the residual  $F(x)$

## 2.4 Automated Planning

Automated planning, also known as AI planning, is the study of how to automatically choose sequences of actions that are anticipated to lead to a desired situation. This is useful for autonomous **agent** that operates in an complex **environment** where there are many possible outcomes that can happen, depending on what the agent chooses to do.

Together, the agent and the environment form the **world**. For a given time  $t$ , the situation of the agent within the environment is called the **state** of the world,  $S_t$ . The initial state is the state from which the plan will start, and a goal state is a state in which the plan will try to end up at. The agent estimates the state of the world through the **observations** it makes,  $O_t$ . The agent is then able to interact with the environment by performing an **action**,  $A_t$ , that is available to the agent at time  $t$ . The state at time  $t$  depends on the previous states of the world, as well as the actions that have been taken.

An automated planner keeps a high-level overview of the environment and creates a sequence of actions, a **plan**,  $\pi$ , that should lead the agent to a goal state.

### 2.4.1 Defining the Planning Problem

The automated planning tasks presented in this chapter are **domain-independent**. That means that they are general enough to solve any planning problem, as long as the environment, available actions, initial state and goal states are given.

To simplify the planning problem, the general planning problem will be restricted by several rules. These rules reduce the planning problem into the **classical planning problem**:

- Time is discretized to a series of time steps:  $t = 0, 1, 2, \dots$
- Actions have no duration.
- Only one action can be performed at a time.
- There is only one agent.
- There are a finite number of states and actions.
- States are fully observable:  $O_t = S_t$ .
- The dynamics of the world is deterministic: An action in a given state will always lead to the same change of state.

These constraints simplify the planning. What is necessary to specify the planning problem is simply:

- $\mathcal{S}$ : The set of all possible states of the world.
- $\mathcal{A}(s)$ : The set of all possible actions applicable at state  $s$ .
- $\Gamma(s'|s, a)$ : The state transition function.

$\Gamma(s'|s, a)$  describes the dynamics of the world. By doing action  $a$  in state  $s$ , the agent and the environment will end up in the state  $s'$  at the next time step.

## 2.4.2 Planning Languages

A planning language is needed to model the world and its dynamics and for the automated planner to understand this model and make plans within it. The following three commonly used solutions will be introduced in this section and their differences presented: STRIPS, partial-order planning and task networks.

### STRIPS

STRIPS was developed in 1971, and can be said to have laid the foundation for modern planning [10]. It was designed to be used for a robot navigating and solving puzzles in complex environments. Using only first-order logic it is a simple language, yet shown to be as expressive as other state-of-the-art planning languages in practice [24]. In STRIPS, states are represented by predicates that describe the objects in the domain. In a given state, predicates give information on all objects in the domain. An action can be applied to a state if the preconditions of the action matches the current state of the world, and when the action is applied it will change the state of the world according to the effects of the action. Searching for a plan in STRIPS is either a search from the initial state towards the goal state, or a backwards search from the goal state towards the initial state. STRIPS is covered in detail in Section 2.4.3.

### Partial-Order Planning

Partial-order planning takes a different approach than STRIPS planning, in that it does not plan from an initial state to the end state. Instead, in partial-order planning the states

themselves are partial plans, and the search consists of combining the partial plans through actions, causal links, variable bindings and ordering constraints. A partial-order plan solution does not necessarily need to be totally ordered such as a STRIPS solution. If it does not matter which one of several actions is performed first, this can be represented by a partial-order plan. Partial-order planning is also a fast form of planning, as shown by Barrett and Weld [3]. However, partial-order planning search algorithms are complex, and they cannot utilize guided searches through heuristics, which will be presented in Section 2.4.4.

### Task Networks

Task networks such as HTN, hierarchical task networks, take a slightly different approach. Instead of searching for a plan that leads to a given goal state, HTNs solves abstract tasks. It takes the abstract task and breaks it down into smaller and smaller sub-tasks until the sub-tasks are at the same level as STRIPS actions. By breaking down the abstract task into a sequence of actions, the plan is created. It can be argued that this type of planning is fitting for many robotic problems, as robots often need to perform certain high-level tasks that can be decomposed into many simple tasks [23].

### 2.4.3 The STRIPS Planning Language

As STRIPS was found to be suitable for the tasks solved in this thesis, the focus of further planning theory lies on tools related to the STRIPS planning language.

Objects, predicates, states and operators are important elements of STRIPS planning. An **object** is an item that is needed in the model of the planning domain. Examples of objects can be "blocks", "doors" and "locations". **Predicates** describe the properties of the objects, and they are either true or false. Examples of predicates can be "block A is on top of block B", "door D is open" and "block B is at location L". A **state**  $s$  in STRIPS is a set of predicates that describe specific objects, also called a set of ground atoms. The state is fully described because all of the ground atoms in  $s$  describe what is true in the state, while everything that is not in  $s$  is false. This way, to check if  $s$  is a goal state,  $s = g$ , it is simply necessary to check if all true statements in  $g$  are in  $s$  while false statements are not in  $s$ . There might be several goal states in the planning problem, but usually we only want to find a plan to reach one of them, ideally the shortest path to any of the goal states. An **operator** in the STRIPS planning domain consists of preconditions and effects. To apply an operator in a state is to perform an action,  $a$ . An example of an action can be "pick up box B at location L". To perform

an action all of its preconditions need to be in agreement with the current state. The AI planning course at the University of Edinburgh [47] formulates it mathematically:

$$(\text{precond}^+(a) \subseteq s) \cap (\text{precond}^-(a) \cap s = \emptyset) \quad (2.25)$$

where  $\text{precond}^+$  is the set of ground actions to be true and  $\text{precond}^-$  is the set of ground actions to be false. If the preconditions of  $a$  are in agreement with  $s$ , then  $a$  is applicable in  $s$ . The state transition that is caused by doing  $a$  in state  $s$  is defined by the effects of the action:

$$\Gamma(s, a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a) \quad (2.26)$$

It can be seen in Eq. (2.26) that the negative effects of  $a$  are deleted from  $s$  and the positive effects of  $a$  thereafter added. If a plan  $\pi = \langle a_1, a_2, \dots, a_k \rangle$  is so that  $\Gamma(s_i, \pi)$  satisfies a goal state for the initial state  $s_i$ , then  $\pi$  is a solution for the planning problem. In Section 2.4.4 the method of searching for such a plan is presented.

## PDDL

For a human to write down a planning problem for the STRIPS planner, a standard for how to formulate the planning problem is needed. The Planning Domain Definition Language, PDDL, is one such standard. PDDL was released in 1998 [28]. It was made in order to create a standard for automated planning languages, so that different planning algorithms could easily be compared and evaluated.

PDDL contains two parts: The **domain description** and the **problem description**. An example of what they look like is given in Fig. 2.10. The domain description is a general description of the object types and the actions available in the planning problem. It specifies the object types and the predicates that describe them. For the action descriptions, the relevant object types, preconditions and effects are given for each action. In the problem description, the specific search problem is written: It contains a list of all objects available, the initial state  $s_i$  and the goal state  $g$ .

### 2.4.4 Search Algorithms

The STRIPS planning problem as presented in Section 2.4.3 can be modeled as a search tree. Each node of the tree is a world state  $s$ , and actions  $a$  are the edges of the tree leading from one state to the next. It will be assumed in this chapter that each action



<pre>(define (domain FRUITS)    (:requirements :strips :typing)   (:types fruit     bowl)    (:predicates     (ontable ?f - fruit)     (empty ?b - bowl)     (inbowl ?f - fruit ?b - bowl))    (:action insert     :parameters       (?f - fruit ?b - bowl)     :precondition       (and (empty ?b)         (ontable ?f))     :effect       (and (inbowl ?f ?b)         (not(ontable ?f))         (not(empty ?b))))    (:action remove     :parameters       (?f - fruit ?b - bowl)     :precondition       (inbowl ?f ?b)     :effect       (and (not(inbowl ?f ?b))         (ontable ?f)         (empty ?b)))  )</pre>	<pre>(define (problem FRUITS-1)    (:domain FRUIT)   (:objects     fruitbowl - bowl     apple - fruit)    (:init     (empty fruitbowl)     (ontable apple))    (:goal     (inbowl apple fruitbowl))  )</pre>
--	--

**Figure 2.10:** PDDL domain and problem specification example. The domain specification, to the left, defines the objects that can be found in the environment, in this case *fruit* and *bowl*, as well as predicates and available actions. The problem specification, to the right, specifies the specific objects found for a single problem instance, as well as a description of the initial state and goal state.

has a cost of 1. This means that the path from the initial state  $s_i$  to the goal state  $g$  at the lowest depth in the tree is the optimal plan.

It is possible to use a simple search algorithms such as BFS, breadth-first search, for searching the tree. The BFS search will guarantee that the optimal plan is found because it fully investigates every depth level of the tree. This is why it is preferred

to DFS, depth-first search, which might find a sub-optimal plan. The disadvantage of BFS, however, is that expanding all nodes at every depth level is computationally expensive.

Using **heuristics** to guide the search, the optimal plan can be guaranteed with a faster algorithm than BFS. A heuristic is an estimated cost of the optimal path from a given node  $n$  to the goal state  $g$ . The cost is denoted  $h(n)$ . The heuristic is **admissible** if it always underestimates this cost. If the estimated cost of every node at depth  $d + 1$  is lower or equal to the estimated cost of its parent node at depth  $d$ , added to the cost of moving to it from its parent node, then the heuristic is **complete**. Formally completeness is given by

$$h(n_{d+1}) \leq h(n_d) + c(n_{d+1}, n_d) \quad (2.27)$$

where  $c(n_{d+1}, n_d)$  is the cost of moving between the nodes.

Two algorithms that utilize heuristics to guide the search are the **greedy best-first search**, GBF search, and the **A\* search**. Greedy best-first search always expands the node with the lowest estimated cost  $f$ :

$$f(n) = h(n) \quad (2.28)$$

The GBF search will typically expand few nodes and find a plan quickly. If  $h(n)$  is closely estimating the actual cost, the greedy best-first search might find a good plan. However, there is no guarantee that this plan is optimal.

The A\* search can guarantee that it will find the optimal plan if the heuristic is admissible and complete. A\* search expands the node that minimizes the following function for  $f$ :

$$f(n) = h(n) + g(n) \quad (2.29)$$

where  $h(n)$  is an admissible and complete heuristic, and  $g(n)$  is the cost to reach node  $n$  from the initial node.

### 2.4.5 Heuristics

The final piece that remains to improve the search algorithms for STRIPS is the design of good heuristics. We can do this by creating a relaxed problem that we can solve instead of the original search problem. A relaxed problem is an problem that is

computationally easier to solve than the original problem, yet provides valuable information about the original problem. To create a relaxed problem of the original STRIPS planning problem, we model the problem by the **Graphplan** algorithm [6].

### Graphplan

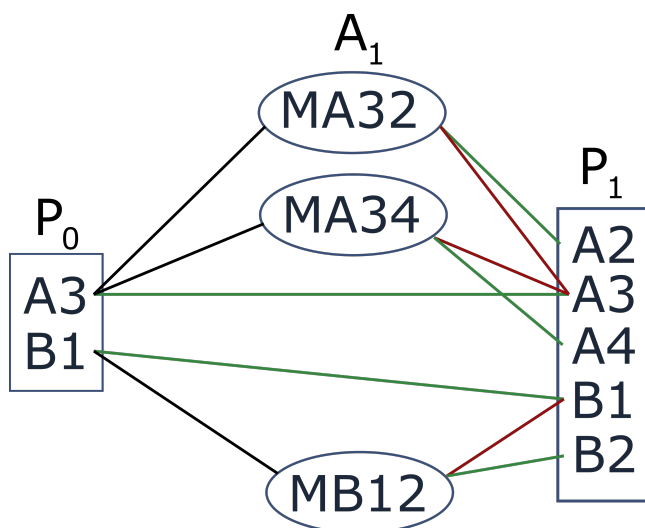
Graphplan designs a planning graph structure that is divided into layers. The first layer describes the initial state  $s_i$  through a set of **propositions**. A proposition is similar to the ground atoms, representing the facts that are true in the current state. An example of a proposition is  $B1$ , which can represent that box  $B$  is at location 1. The second layer describes the actions that can be applied in the initial state, resulting in the third layer which contains all propositions that are effects of the actions of the previous layer. The planning graph continues to expand, every even numbered layer being actions and odd numbered layer being state propositions. A simple planning graph is shown in Fig. 2.11.

When using Graphplan to find heuristics, we use a relaxed version of the planning graph: The negative effects of the actions are ignored. For instance, a block  $B$  that is moved from location 1 to location 2 will afterwards be located at both location 1 and 2. This simplifies the search through the planning graph significantly.

### Planning Graph Heuristics

The heuristics presented in this section are calculated for every node that is found while searching through the search tree described in Section 2.4.4. Before expanding a new node in the tree, all of the candidate nodes on the fringe of the already explored nodes turn into a planning graph with that particular node as its initial state. For this reason it is important that the heuristic that used is fast.

The heuristic functions  $hMax$  and  $hAdd$  are similar, and they are found by searching through the relaxed Graphplan. Both search until all of the propositions in the goal state are achieved.  **$hMax$**  returns the number of actions it took for all propositions to be achieved, ie. the number of actions it took until the most expensive proposition was made true. It is the minimum number of actions needed to reach the goal state, and it is therefore an admissible heuristic. If the actions needed to achieve the most expensive goal proposition by chance achieve all other goal propositions after the final action, then the estimate would be correct. As this is rather unlikely,  **$hAdd$**  instead adds up the number of actions needed to achieve every single goal proposition. This might be an overestimation, and the heuristic is therefore not admissible.



**Figure 2.11:** A simple planning graph. In this world there are two boxes,  $A$  and  $B$ , that can be moved to adjacent locations. The initial position of the boxes is described in  $P_0$ . Depending on which action that are chosen in later  $A_1$ , there are many possible states the boxes can end up in at  $P_1$ . Red lines indicate negative effects, and green lines indicate positive effects: If box  $B$  is moved from location 1 to location 2 by the action  $MB12$ , then the red line indicates that  $B$  no longer is at location 1, whereas the green line indicates that  $B$  is at location 2. The planning graph is as expressive as STRIPS.

The **FF heuristic** also searches through the relaxed Graphplan, but in contrast to hMax and hAdd it actually tries to solve the relaxed planning problem [18]. For the heuristic to quickly find a solution, it does not find the optimal plan of the relaxed problem, but rather finds an approximate plan quickly. Therefore, although the FF heuristic is not admissible, it finds an accurate heuristic without being computationally expensive.

## 2.4.6 Planning in a Real-World Environment

For an agent operating in a real-world environment, the rules of the classical planning problem that was introduced in Section 2.4.1 might no longer hold. Some of the important adjustments to the rules in a real-world environment typically include:

- States are only **partially observable**. The observations are limited by the sensor of the agent. Some objects may be out of sight because they are blocked by other objects or because they are located in a different area than the agent.
- The dynamics of the world are **non-deterministic**. An action in a given state may be performed incorrectly, or the effect of the action may be different from our expectation.
- **Unexpected events** may occur. For example, objects may fall over, roll away or be displaced by humans while the agent is operating in the environment.

The challenges of a real-world environment require a flexible planner that can adjust to new information. One solution, as presented by Gallab et al. [13], is the **Run-Lookahead** algorithm:

```

1: procedure RUN-LOOKAHEAD( $\Gamma, o, g$ )
2:    $s \leftarrow$  state estimation based on  $o$ 
3:   while  $s \neq g$  do
4:      $\pi \leftarrow$  lookahead( $\Gamma, s, g$ )
5:     if  $\pi$  fails to find a plan then
6:       return failure
7:     end if
8:      $a \leftarrow$  first action of  $\pi$ 
9:     Go one step by performing  $a$ 
10:     $o \leftarrow$  observation
11:     $s \leftarrow$  state estimation based on  $o$ 
12:  end while
13: end procedure

```

**Algorithm 2.2:** Run-Lookahead

Algorithm 2.2 attempts to solve the problems of partial observability, non-determinism and unexpected events. In Line 2 and Line 11,  $s$  represents an estimation of the current state, based on the sensor input of the agent. This handles the problem of partial observability. State variables of  $s$  that are not observable may be estimated based on their state when previously being observed in combination with a model of the environment and the state transition function  $\Gamma(s'|s, a)$ . Non-determinism and unexpected events are handled as follows: After every action that the agent carries out in Line 9, the agent makes a new observation of the environments and creates a new plan in Line 4. Only the first step of the plan is executed before the agent replans by the *lookahead* function.



# Chapter 3

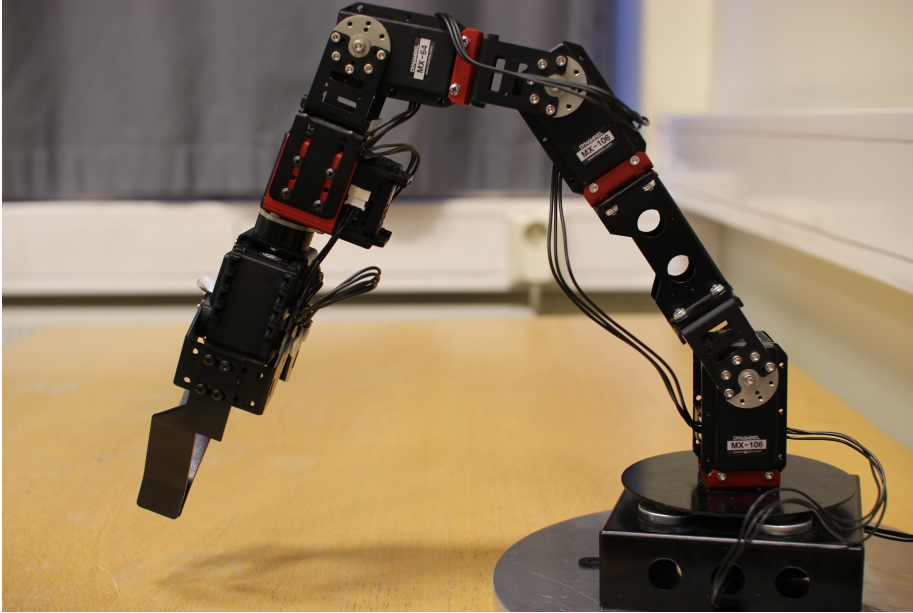
## Tools and Libraries

This chapter presents the robotic manipulator and all software tools and libraries that were used in this thesis. Everything was implemented in Python 3, so the data tools presented in this chapter provide a useful programming interface for Python 3, with the exception of the **libfreenect** toolbox described in section 3.4.

### 3.1 Robotic Manipulator

The robotic manipulator was custom built using actuators from the company ROBOTIS. The South Korean-based company develops humanoid robots for education purposes, and their focus is on commercializing personal robots [22]. However, they also create actuators and software for robotics in a wide range of fields. Their Dynamixel actuators and robots have been used, among others, for medical and rescue purposes, for surveillance and in the military.

The full robot can be seen in **Fig. 3.1**. It consists of seven Dynamixel actuators that are linked together in a rigid metallic structure. A wire system running along the robotic arm provides power to every actuator from only one single power supply. In addition, the wire system allows for reading and writing to every actuator by the use of a small device called USB2Dynamixel. The hardware of the manipulator is presented in detail in this chapter, and the setup of the robot will be presented in subsection 4.2.1.



**Figure 3.1:** Side view of the manipulator. From the rotating base all the way until the tip of the grippers, the robot measures 57 cm.

### 3.1.1 Dynamixel Actuators

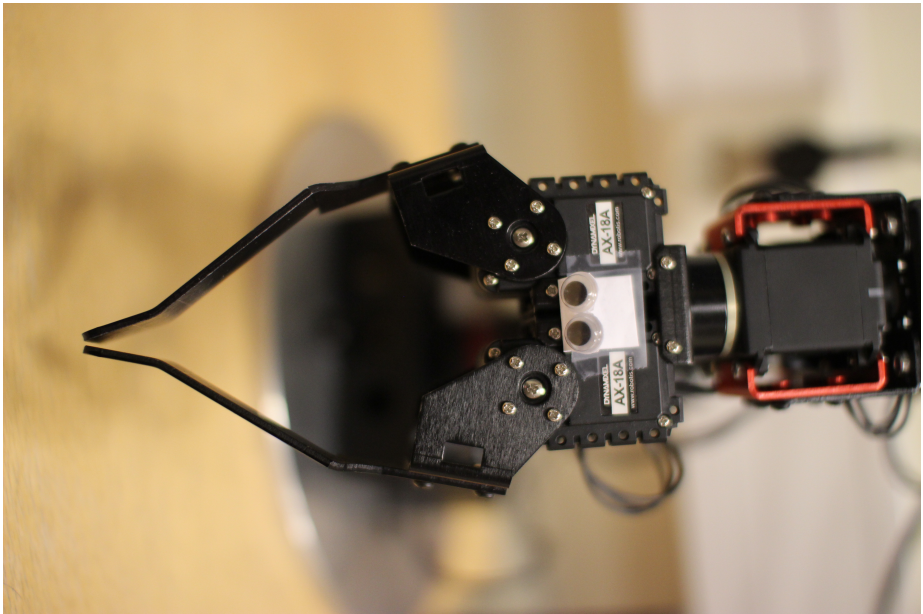
There are seven Dynamixel actuators. All of the actuators are revolute. Their model number as well as each actuator's purpose for the motion of the manipulator will be presented. The full details of each actuator model can be found in the datasheet that is referenced after the actuator model name is presented. Starting from the base of the manipulator and following the robotic arm upwards towards the grippers on the manipulator's wrist, the details of the seven motors are as follows:

- **MX-28 [44] waist actuator:** This actuator is mounted below the circular base of the robotic arm. The waist can spin freely  $360^\circ$ .
- **MX-106 [43] shoulder actuator:** This actuator is used to raise and lower the upper arm of the robot.
- **MX-106 [43] upper elbow actuator:** This is the actuator at the middle of the



robotic arm, used to raise and lower the middle arm.

- **MX-64 [45] lower elbow actuator:** Similarly to the the MX-106 actuators, this is a joint that is used to lift and lower the forearm of the manipulator. It is the last joint of the arm of the manipulator.
- **MX-64 wrist rotating actuator:** This actuator allows the robot hand to spin freely 360°.
- **Two AX-18A [41] gripper actuators:** These actuators each control the position of one of the grippers on the wrist. By using them synchronously, the manipulator can perform actions of gripping and releasing objects. The grippers can be seen in **Fig. 3.2**.



**Figure 3.2:** The grippers

### 3.1.2 Programming Interface

The USB2Dynamixel [42] is a device that is used for communication between the computer and the Dynamixel actuators. The actuators in this custom built manipulator use a 3-pin connector system and is compatible with TTL communication.

By writing to and reading the actuator registers, it is possible to control the manipulator through programming software. ROBOTIS has developed the free software program RoboPlus [40], in which this is done automatically: Through this software it is possible to setup the Dynamixel actuators by using a program called Dynamixel Wizard. ROBOTIS has also developed an SDK with C functions for writing and reading to Dynamixel actuator registers. Included in the SDK, ROBOTIS has provided a Python wrapper so that one can write Python code for reading and writing to the robot.

## 3.2 IKPy and URDF

IKPy is a software toolbox that allows the user to compute the inverse kinematics of any modelled robot [26]. The robot model is imported from an URDF (Unified Robotic Description Format) file that specifies the position of each actuator and the limits the every actuator's movement. The IKPy software can visualize the robot joints so that the robot can be virtually simulated.

In the thesis, IKPy was used in order to find the joint positions of the robotic manipulator when specifying which location the gripper should be at.

## 3.3 Kinect

The Kinect was developed by Microsoft to be used with the Xbox 360 video game console. It consists of a standard RGB camera, a depth camera, an accelerometer and a microphone. Although it was initially meant to be used for video games, open source toolboxes such as Libfreenect [32] that can access its sensors have made it a useful and cheap tool for robotic purposes. An analysis of the Kinect's data quality was performed by Mankoff and Russo [27], presenting the high accuracy and precision in the depth images of the device. They also utilized the Kinect to track the surface height of glaciers, showing the broad application possibilities of the Kinect within robotics.

An image of the Kinect is seen in **Fig. 3.3**. The RGB camera takes images at a



**Figure 3.3:** The Kinect. The three main sensors are visible behind circular glass covers. Left: IR light projector. Middle left: RGB camera. Middle right: IR light detector (depth camera).

resolution of 640x480 pixels. There is an IR (infrared) light projector and an IR light detector that together give depth information. The projector emits infrared lights in a specific pattern. The IR detector, also called the depth camera, thereafter records the IR light that bounces off of objects in front of the Kinect. By comparing the pattern of the reflected IR light to a previously known pattern, the depth camera finds the distance to each pixel in the image [27]. The depth camera also has a resolution of 640x480 pixels.

The Kinect was used in the perception part of the system, acting as "eyes" for the robotic manipulator.

## 3.4 Libfreenect and Cython

Libfreenect is a software tool that allows for reading of the Kinect's sensors [32]. The software is open source and maintained by the OpenKinect community. Not only does

Libfreenect cover simple functions for reading RGB and depth images taken by the Kinect, but the software also contains functions that can be used as a basis for transformations between the different camera image planes as well as transformation from the image plane to 3D space.

The Libfreenect software is mainly written in the C language. There is a workaround for this problem: By using Cython [4], it is possible to write Python code that can translate between Python and C code at any point so that C functions can be used. A simple wrapper using Cython to call C functions from Python was included in the Libfreenect software. However, this wrapper only included support for very the simple functions of reading the RGB and depth images of the Kinect. Therefore, when adding additional functionality using internal Libfreenect information, such as Kinect camera configurations, the code had to be written in C and thereafter be loaded into Python through a tedious process that was complex and time-consuming.

In this project, Libfreenect was used in order to read the Kinect sensors as well as for performing transformations related to the intrinsic parameters of the Kinect.

## 3.5 OpenCV

OpenCV, the Open Source Computer Vision Library, is an open source software library for use on image and video data [20]. It provides highly optimized code for simple camera vision tasks, so that users of the library can focus on more advanced tasks and research.

OpenCV was in this thesis used to perform a number of operations on the images taken by the Kinect.

## 3.6 YOLOv3

YOLOv3 is a real-time, state of the art system for detecting objects in images and video [39]. The letters of the system is an acronym for "You Only Look Once", which is describing for its design: It analyses an image by reading the image only once in order to feed it into a single neural network. YOLOv3 is an improved version of the YOLOv2 [38] and its predecessor YOLO [37]. YOLOv3 is very fast, analyzing images at more than 30 frames per second while still reaching state of the art precision.

YOLOv3 was used in this thesis to identify objects within images taken by the

Kinect, so that they could be isolated and localized in 3D space.

### 3.6.1 Darknet-53

Darknet-53 is an open source implementation in C of a deep residual convolutional network, and it forms the basis of the YOLOv3 system [36]. It consists of 53 convolutional layers, as the name suggests, as well as a number of residual layers. At the end of its pipeline, global averaging filters followed by a fully connected layer is fed into a softmax function. All of these concepts were introduced in section 2.3.

### 3.6.2 Architecture

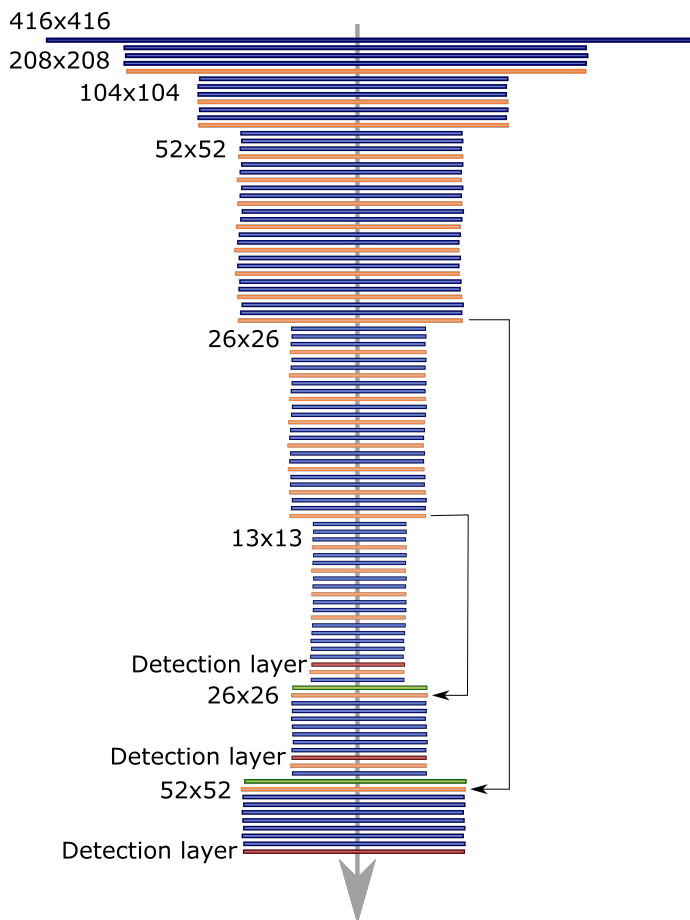
YOLOv3 is built upon the Darknet-53, however many more layers are added. The final design contains a total of 106 layers. The full network can be seen in Fig. 3.4.

#### Predictions

YOLOv3 takes an image as the input and outputs a prediction of what objects can be seen in the image. Not only does it predict the objects that are present, but YOLOv3 also specifies a rectangular box, a **bounding box**, for each object that should be just big enough to fully contain the object. A bounding box prediction for one object consists of 5 variables:  $x, y, w, h$ , describing the position of the bounding box, in addition to the prediction confidence. In YOLOv3, the image is divided into an  $S \times S$  grid. Each grid cell is allowed to predict  $B$  bounding boxes. There are  $C$  number of possible classes that an object can belong to. The output contains a total of  $S * S * (5B + C)$  number of values: Each cell has predicted  $5B$  values that specify the 5 bounding boxes, and all  $C$  number of classes are evaluated to determine to how likely it is that the objects within each bounding box belong to those specific  $C$  classes.

#### Anchor Boxes

The bounding boxes predicted by YOLOv3 are not predicted from scratch. They are predicted as offsets from a set of 9 **anchor boxes**. The anchor boxes are pregenerated boxes that are found through k-means clustering of the true bounding boxes in the training data. Using the anchor boxes as the basis for prediction, each cell is in fact



**Figure 3.4:** All 106 layers of the YOLOv3 architecture are visualized, along with the size of the output data of each layer. Blue colored layers are convolutional layers. The orange colored layers are residual blocks, into which the previous layer and a forwarded layer from earlier are inserted as shown in Fig. 2.9. Two of the residual layers have an especially long feed-forward effect, and these effects were therefore marked by arrows. The two green colored layers are upscaling layers. Finally, the three red colored layers are detection layers, also labeled by text.

predicting coordinates that describe manipulations of the anchor boxes such that the result becomes bounding boxes around objects.

### Detection Layers

In order to detect small objects as well as large one, YOLOv3 predicts bounding boxes at three different layer sizes. This is done by upscaling two times and performing predictions at the three detection layers as seen in Fig. 3.4.

Because the layers are different in size, the grid size  $S$  is different at every layer. Moreover, the 9 anchor boxes are distributed between the detection layers so that the smallest detection layer uses the 3 biggest anchor boxes, the medium sized layer uses the 3 medium sized anchor boxes, and the largest layer uses the 3 smallest anchor boxes. This way, the smallest detection layer is responsible for detecting the largest objects whereas the biggest detection layer detects the smallest objects.

Predicting at three levels increases accuracy greatly, although it results in ten times the amount of predicted boxes as the predecessor system YOLOv2 predicts. For this reason, YOLOv2 was significantly faster, analyzing images at 40 FPS. YOLOv3 sacrificed some of the speed in order to improve its accuracy.

### 3.6.3 Python Interface

The Darknet implementation is written in C. However, it is possible to call upon functions in the C language from Python by using the "ctypes function library". The framework for using Python and loading OpenCV images into the detection function was provided within the Darknet implementation. Only small changes had to be made to the framework, and object detection could easily be integrated into the manipulator system.

## 3.7 Pyntcloud

Pyntcloud [9] is a library in Python for visualizing and working with point clouds. When using Pyntcloud in Jupyter Notebook, Pyntcloud can take a matrix that describes the location of points in 3D, as well as a color associated with each point, and visualize it in 3D.

Using Pyntcloud, 3D visualization of the data points found the Kinect's cameras was performed.

## 3.8 Pyperplan

Pyperplan is a STRIPS planner implementation in Python developed at Albert Ludwigs University of Freiburg in 2010 and 2011 [2]. It can parse text files in the PDDL format that specify the planning domain and problem. Pyperplan turns the PDDL planning domain and problem into a search tree, in which all reachable states are represented as nodes.

Although the standard search algorithm in Pyperplan is a breadth-first search algorithm, a wide range of more sophisticated heuristic search algorithms have been implemented. For example, the FF heuristic presented in subsection 2.4.5, as well as the A\* search and the greedy-best-first search algorithms described in subsection 2.4.4, are implemented in Pyperplan. Combinations of these can be used to provide a good guidance for the search.

In this thesis, Pyperplan was used for creating plans for the robotic manipulator.



# Chapter 4

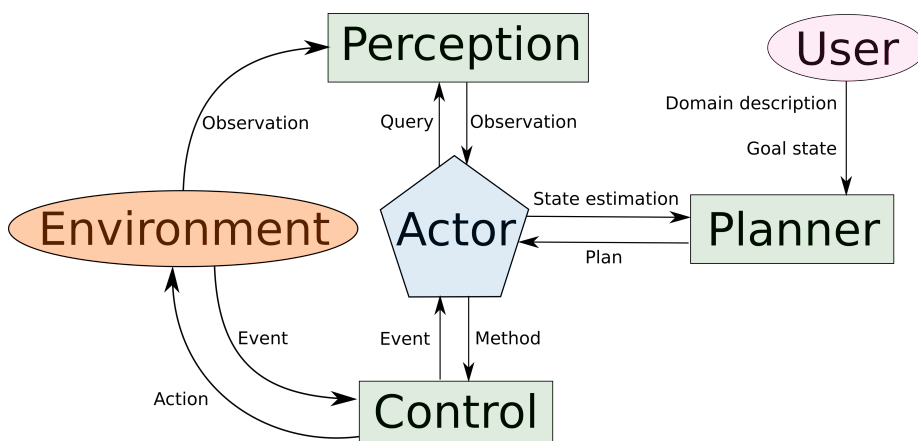
## Design and Implementation

In this chapter, the process of designing and implementing the system is described in detail. In Section 4.1, the system design and how it was divided into a number of **modules** is explained. The implementation of these individual modules is presented in Section 4.2, Section 4.3 and Section 4.4. As it is intended that the system will be used in future research, the set-up and implementation of the robot is explained in great detail.

### 4.1 System Design

The software implementation of the robot manipulator should be able to perform several tasks, ranging from high-level intelligence planning tasks to low-level robot control. The system was therefore separated into separate modules, each module responsible for a single set of tasks: The **control module** was responsible for controlling the robot. This included the configurations of all motors, reading and writing to the actuators, and calculations of the location and angle of the robot joints. In short, all tasks directly related to the movement of the actuators were handled by the control module. The **perception module** was responsible observing of the environment, acting as the eyes of the robot. Finally, the **planning module** was responsible for automatically creating high-level plans for the robot to carry out.

An **actor** was responsible for organizing the information from the three modules and act accordingly. Not only does an actor need to act out the plan from the planner, but in a real world environment, an actor needs to react to and handle imperfections such as unforeseen events, insufficient observations, and actions that do not have the expected effects as well.



**Figure 4.1:** An overview of the system design. The information flow is visualized with keywords explaining the nature of the communication between the modules. The design was partly inspired by Nau et al. [29].

An overview of the system design is given in Fig. 4.1. The user-specified goal state may be a goal that is specified beforehand, or it may be given during runtime. Regardless, this is specified by the user of the program, and it is the only intervention made by the user: The rest of the system should be fully automatic. The environment gives information to the system in two ways: Both through observations made by the camera in the perception module, as well as directly to the control module. The control module receives information from the environment through response of the robotic movement. For example, objects might feel hard or soft when gripping them.

## 4.2 The Control Module

This section describes the control module, which is responsible for physically controlling the manipulator's joints. For the robotic manipulator in this thesis, the role of the

controller module was to steer the grippers to any given position in the vicinity of the robot in a safe manner and at a reasonable speed.

In Section 4.2.1 the set-up of the manipulator is explained. Section 4.2.2 explains how a model of the physical robot was created. Finally, solving the inverse kinematics problem for the robot is presented Section 4.2.1.

### 4.2.1 Set-up of the Manipulator

Each actuator in the robotic manipulator was individually connected to a computer through the USB2Dynamixel device, and the RoboPlus software [40] confirmed that everything worked by reading and writing to registers in the actuators. The seven actuators were each given a unique ID number from 1 to 7. The baud rate of the communication with all actuators was set to 1000000 bits per second. The ID and baud rate settings are stored in the permanent EEPROM memory of the actuators. After performing these steps, the actuators were connected in a daisy chain, so that it was possible to communicate with each actuator individually using only a single connection between the USB2Dynamixel device and the base actuator.

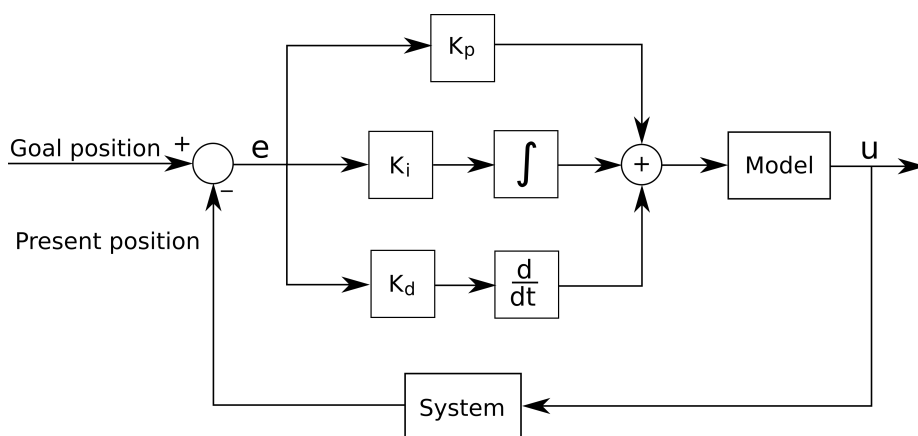
As the arm of the robotic manipulator was heavy, it fell every time the arm was lifted up from the ground. Therefore, a custom made circular metal plate with a diameter of 24 cm was built for the robot to stand on. Three bolts were used to fasten the robot to this metallic plate.

Using the Python wrapper developed by ROBOTIS, it was simple to read information from the joints actuators. Information on the current position, speed and load on each actuator could be found by reading from the RAM memory of the actuators. To control each actuator, specific RAM memory addresses in the actuators are written to. Some of the useful read/write commands include:

- **Present position:** Read only. Returns the current position of the actuator. The position can range from  $0^\circ$  to  $360^\circ$ .
- **Goal position:** Read/Write. This is the reference position of the actuator.
- **PID:** Read/Write. These are three registers that specify the  $K_p$ ,  $K_i$  and  $K_d$  gains of the inbuilt PID regulators.
- **Present speed:** Read only. Returns the current rotational velocity of the actuator. The range of the speed can be in the range 0 - 937 RPM either clockwise or anticlockwise.

- **Speed limit:** Read/Write. Maximum rotational velocity.
- **Present load:** Read only. Returns the current load on the actuator.
- **Torque limit:** Read/Write. Maximum torque allowed to be applied by an actuator.

### Adjusting the PID Controllers



**Figure 4.2:** An overview of the PID controller inside each actuator. The output of the model,  $u$ , is the current that is applied to the actuator.

Every Dynamixel actuator in the robotic manipulator arm has an inbuilt PID regulator. The PID regulators are active at all times when the robot is powered on, keeping the joints at the desired angles. The PID controller is illustrated in Fig. 4.2. The system in Fig. 4.2 can be represented mathematically:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt} \quad (4.1)$$

where  $e(t)$  is the error. The error is the difference between the reference and measured value. The proportional term,  $P = K_p e(t)$ , is simple a gain that is proportional to the difference at the current time. A larger  $K_p$  allows for faster regulation, but may lead to overshooting and an unstable system. The integral term,  $I = K_i \int_0^t e(\tau) d\tau$  builds up

the error over time. This way the system is brought to a state of zero error after some time even if there are disturbances in the system or if the model is slightly wrong. Finally, the derivative term  $D = K_d \frac{de(t)}{dt}$  controls on the rate of change of the error, effectively dampening quick changes in the error.

The PID controller in each actuator takes two inputs: The reference "Goal position", and the measured "Present position", both read from the RAM memory. The PID controllers use a discretized variant of Eq. (4.1) to calculate a suitable control variable  $u(t)$  at every time step. The control variable corresponds to the current that is applied to the actuator.

The PID gains were adjusted so that the arm of the manipulator quickly and steadily could be moved to any position by simply changing the goal position of each actuator. Without any integral gain, a simple proportional regulator was too weak to counteract the gravitational force acting on the arm. It was found that the derivative term was unnecessary for these operations. The derivative term is sensitive to measurement noise, and might be omitted if the system is fine without it, so it was set to 0. When tuning the regulators, suitable values for the PID control were found to be:

$$\begin{aligned}K_p &= 4.0 \\K_i &= 4.9 \\K_d &= 0\end{aligned}\tag{4.2}$$

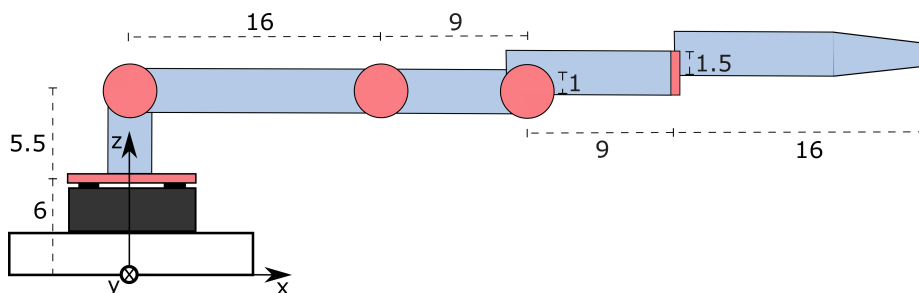
### **Controlling the Grippers**

To grip and hold onto items without pinching too hard, the actuators controlling the grippers needed to have a limit on their torque. After setting this limit, the grippers worked as follows: The grippers open and release by setting the "Goal position" of the two actuators to positions according to the desired opening of the grippers. When an object was located in between the grippers and the "Goal position" of the grippers was set to the closed position, the object was considered to be held when the "Present load" reached the value of the "Torque limit". The grippers tried to reach any given position unless an object was blocking, in which case the grippers would apply all of their power, limited by "Torque limit", to hold onto the object.

## 4.2.2 Modelling the Robot

A Cartesian coordinate system with the origin at the base of the robotic manipulator was created. This coordinate system, referred to as the **Cartesian robot coordinate system**, has its origin directly underneath the metal stand of the manipulator. One unit of this coordinate system was set to represent 1cm in the real world.

The robot was modelled in a URDF file, using the Cartesian robot coordinate system. In the URDF file, the shape of the robotic manipulator was described. Each actuator was specified with the following parameters: Type of joint, distance from the previous actuator, and lower and upper limits on the angles of the joint. To model the robot accurately, a tape measure was used to find the distances between every joint of the manipulator. The limits on the actuator angles were found by bending each joint to its extreme positions. The first actuator is special because it has no limits; it can rotate freely more than  $360^\circ$  in both directions. However, all other joints had limitations on their movement.



**Figure 4.3:** Model of the manipulator with directions and measurements as specified in the URDF file. The joints are colored in red. Measurements between the joints are given in centimeter. The base coordinate system is seen near the base of the robot. The two actuators used to control the grippers are not showed in this figure because they do not influence the inverse kinematic control algorithms.

A model of the robotic manipulator is presented in Fig. 4.3. The model is a visual representation of the information given in the URDF file.

### 4.2.3 Inverse Kinematics

The inverse kinematics problem, described in Section 2.1.2, was used to find the angle of each actuator in order for the grippers to reach a certain point with a given orientation. The desired position and orientation is referred to as the **goal state**,  $g$ , of the grippers. The goal state was specified in a transformation matrix. For example, the transformation matrix of

$$g = \begin{bmatrix} 1 & 0 & 0 & 50 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 14 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

specifies a goal state where the gripper is located at  $d = [50 \ 0 \ 14]^T$  and pointing along the  $x$  axis. This goal state is the state of the grippers in the robot model in Fig. 4.3.

The **final state** of the gripper,  $s'$ , is a function of the angles of all joints in the arm of the robot:  $s(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5)$ . Let  $\theta$  be the set of all angles,  $\theta = \{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5\}$ . Define a function  $f(x)$  as follows:

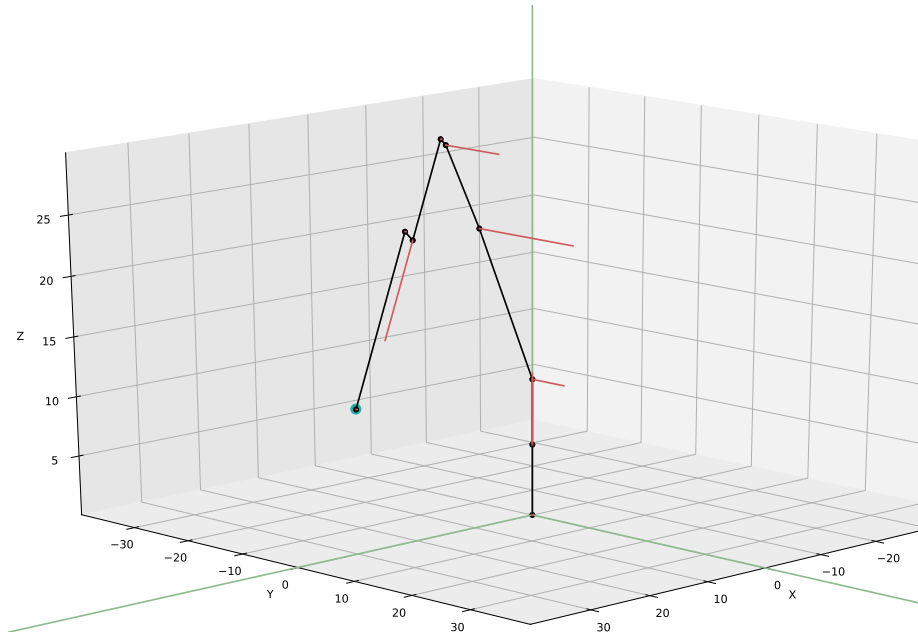
$$f(\theta) = \|s'(\theta) - g\| \quad (4.3)$$

Minimizing Eq. (4.3) is the same as approximating  $s'(\theta)$  to  $g$ . This can be exploited to turn the inverse kinematics problem to an optimization problem. The optimization problem can be formulated as follows:

$$\begin{aligned} \min_{\theta \in \mathbb{R}^5} \quad & f(\theta) \\ \text{subject to} \quad & \theta_{i,min} \leq \theta_i \leq \theta_{i,max} \quad i = 1, 2, \dots, 5 \end{aligned} \quad (4.4)$$

where  $f(\theta)$  is defined in Eq. (4.3). The restrictions of each angle  $\theta_i$  are the limits on the movement of each actuator, as described in Section 4.2.2.

The L-BFGS-B algorithm presented in Section 2.1.3 was already implemented in IKPy, although it was made to solve a simpler problem which did not consider the current state of the grippers nor the goal orientation of the grippers. Modifying the function and applying it to the full optimization problem in Eq. (4.3), the L-BFGS-B used the goal state  $g$ , the constraints of the angles, and the current orientation of the grippers  $\theta_0$  as inputs and could solve the inverse kinematics problem.



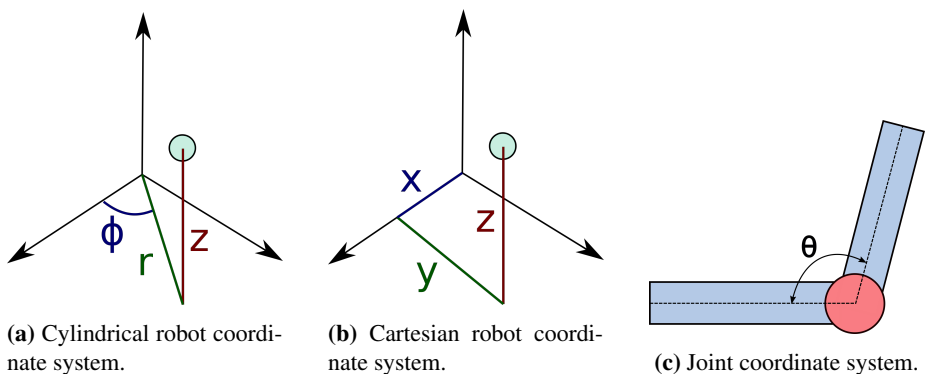
**Figure 4.4:** 3D model of the manipulator. The blue sphere indicates the goal position of the grippers, whereas the robot structure is drawn as a series of black lines. The red vectors pointing out of the robot are the rotational axes about which the five actuators can rotate. It is seen in the model that the robot has reached the goal position of the grippers because the tip of the grippers is located inside the blue goal position sphere.

The output of the L-BFGS-B algorithm is the vector of actuator angles  $\theta_{opt}$  that solves Eq. (4.3). Some goal state configurations might be unreachable, but the algorithm will in that case return a state that is near the goal state. Using the URDF model and specifying the angles with  $\theta_{opt}$ , IKPy modeled the robot in a 3D environment. A 3D model of the robot is seen in Fig. 4.4.



#### 4.2.4 Robotic Control

Three coordinate systems were used for control of the robotic manipulator. For a human who is interacting with the robot, it is natural to describe the position and movement of the robot in a **cylindrical robot coordinate system**. This is because the only horizontal movement of the robot is achieved by rotating its the waist actuator. Therefore it is easy to use that angle to describe positions. Goal positions were given in the cylindrical coordinate system. The cylindrical coordinate system is seen in Fig. 4.5a. Because the inverse kinematics problem used the Cartesian robot coordinate system, the goal coordinates specified in the cylindrical coordinate system had to be converted to Cartesian coordinates, as seen in Fig. 4.5b. The output vector of the L-BFGS-B algorithm specifying the angles of the joints of the robot were given in the **joint coordinate system**. In this coordinate system, angles were restricted between  $0^\circ$  and  $360^\circ$ . An angle of  $180^\circ$  represented a fully stretched joint, whereas an angle of  $0^\circ$  represented a joint folding in on itself. This system can be seen in Fig. 4.5c.

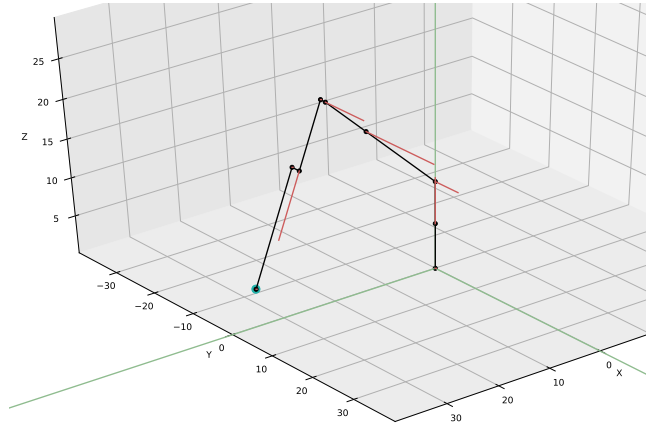


**Figure 4.5:** Three different coordinates systems were used for robotic control.

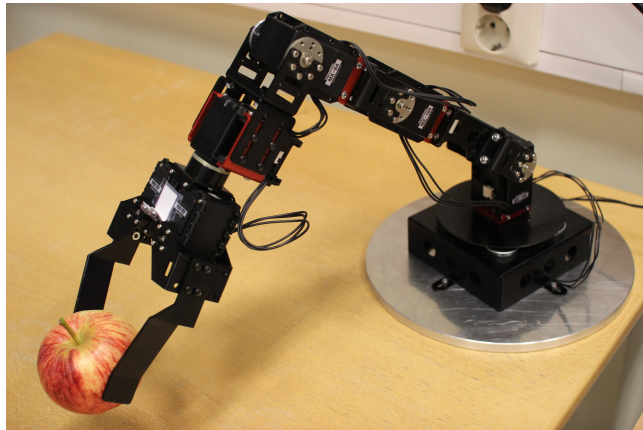
Moving the robot to the goal position specified in the cylindrical robot coordinate system was done by writing the angle of every actuator in the joint coordinate system to the RAM memory of the corresponding actuators. This would activate the PID control of the actuators, causing every actuator to move to and stabilize at the specified angle. The movement between the starting point and the end point of the robot was therefore not controlled, and the robot might accidentally move into obstacles on the way from one location to another. There are solutions to get around the problem. One solution to gain control over the robot movement is to divide great distances into a series of waypoints, guiding the robot small distances at a time. Another solution is to

divide the workspace of the robot into safezones, letting the robot move freely in zones where there are guaranteed to be no obstacles.

Figure 4.6 shows a 3D model of the robot found by using the L-BFGS-B algorithm, as well as the physical robot after moving to the given state. The goal state was specified to be at position  $d = [35 \ 0 \ 5]^T$  with the grippers pointing downwards. Although this goal state, in which the grippers point directly downwards, is unreachable, the robot found the most similar solution.



(a) A plot of the optimal position of the manipulator joints after performing inverse kinematics optimization. The tip of the manipulator is indicated by the blue sphere.



(b) The physical manipulator after moving to the position specified in 4.6a. The manipulator is currently holding an apple.

**Figure 4.6:** A model of the robotic manipulator was firstly found as shown in 4.6a. The positions of each joint were passed to the physical manipulator as goal positions. In 4.6b, the manipulator reached the specified position through PID control. The coordinate system axes in the two figures are approximately lined up, although the origin is displaced.

## 4.3 The Perception Module

The perception module acts as the robotic manipulator's eyes. In order for a robot to behave autonomously, it needs one or more sensors, such as depth camera, that can perceive the world around it.

As the eyes of the robotic manipulator, the perception module needs to locate objects that the manipulator can interact with. The grippers of a robot are small, so localizing objects in 3D space needs to be **accurate**. Moreover, it needs to operate in **real-time**, as the robot needs to react to changes in its working area. These two qualities have been of influence to the design of the perception system presented in this chapter.

This section follows the implementation of the perception module, starting by a short discussion of different camera vision solutions and ending with a working system for locating objects in 3D by the use of a Kinect camera. The steps in this process consisted of the following:

1. Choosing the suitable camera hardware.
2. Object detection.
3. Camera calibration.
4. Determining the three-dimensional position of the detected objects.

### 4.3.1 Choice of Camera Hardware

For a robot that operates in a three-dimensional environment, a single RGB camera is not enough as colored pixels on an image plane is not enough to determine distance alone. There are a number of possible solutions for capturing depth information in images:

- Using several cameras. Before using two or more cameras for depth vision, the cameras can be calibrated by taking images of the same flat surface with a known pattern. Toolboxes such as OpenCV provide readily available solutions for calibration of  $n$  number of cameras. The different cameras must point to the same object, and by using triangulation on the position of the object in the different images, the object's 3D location can be revealed.

Using several normal cameras and triangulation might be an affordable option to determine the three-dimensional location of objects. However, the need for careful measurement of the position of each camera in the room, as well as the need for calibration, adds to the complexity of this method.

- Using a stereo camera. A stereo camera has two or more camera lenses in one single apparatus. As the baseline distance between the lenses is known, calibration of a stereo camera is easier than multiple camera calibration.
- Using an RGB-D camera, ie. a camera together with a separate depth sensor. By using a dedicated device to find the depth information, it is not necessary to perform triangulation to determine location in 3D. However, the camera and the depth sensor need to have their coordinate systems aligned in order to work fluently together.

The Kinect, an RGB-D camera, was chosen to be used for the perception module of this thesis. It is a device that contains a normal RGB camera in addition to a dedicated depth sensor, functioning as described in section 3.3. Kinect is a well-known device used in robotic projects, praised as an affordable yet highly accurate device [27].

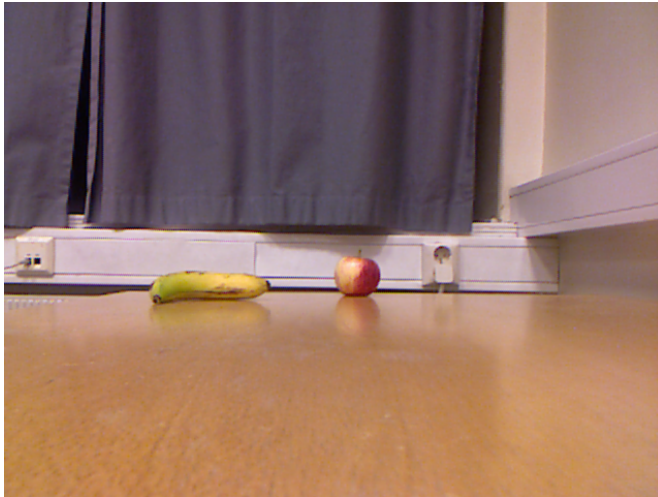
### 4.3.2 The Kinect Sensors

The Kinect has two sensors: The RGB camera and the depth sensor. An image taken by the RGB camera is seen in Fig. 4.7a. A short explanation of the depth camera is given in section 3.3. The depth camera finds the distance to the objects that reflected the IR light that hits the sensor. The Kinect is located in a coordinate system in which the  $z$  axis points out of the depth camera, as seen in Fig. 4.8, so the depth camera finds the  $z$  coordinate of the objects.

Using Libfreenect, the distance to each pixel in the depth image along the  $z$  axis can be given in millimeters. To visualize the depth information given by the depth camera, the following formula was used to convert all depth information points,  $dp$ , to values that can be visualized,  $vp$ :

$$vp = dp / \max(dp) \cdot 255 \quad (4.5)$$

where  $dp$  is a matrix containing every depth points measured in millimeters, and  $vp$  is a matrix in which all depth points have been re-scaled to fall between the values of 0 and 255.  $vp$  is visualized as a black-and-white image in Fig. 4.7b. The lighter

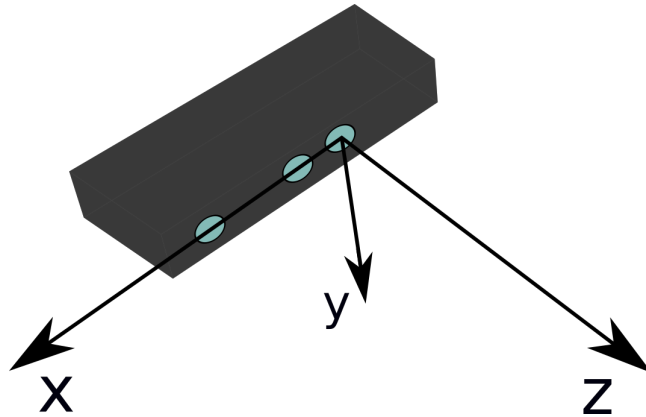


(a) Image taken by the RGB camera.



(b) Visualization of the depth information taken by the depth camera.

**Figure 4.7:** Color and depth images taken by the Kinect.



**Figure 4.8:** The coordinate system of the Kinect camera. The origin of the coordinate system is in the middle of the depth camera.

the pixel, the further away its location is. Black points in the image are areas with no depth information. Objects are too close to the depth sensor lack depth information, as is seen with the closest part of the table in Fig. 4.7b. The nearest distance that can be measured by the depth sensor was found to be 50cm. Areas that are not reached by the IR projected light are also left dark. An example of this effect can be seen on the left side of the fruits as well as in the opening between the curtains in Fig. 4.7b. The effect usually affects the left side of objects because the IR projector is located on the right-hand side of the depth camera.

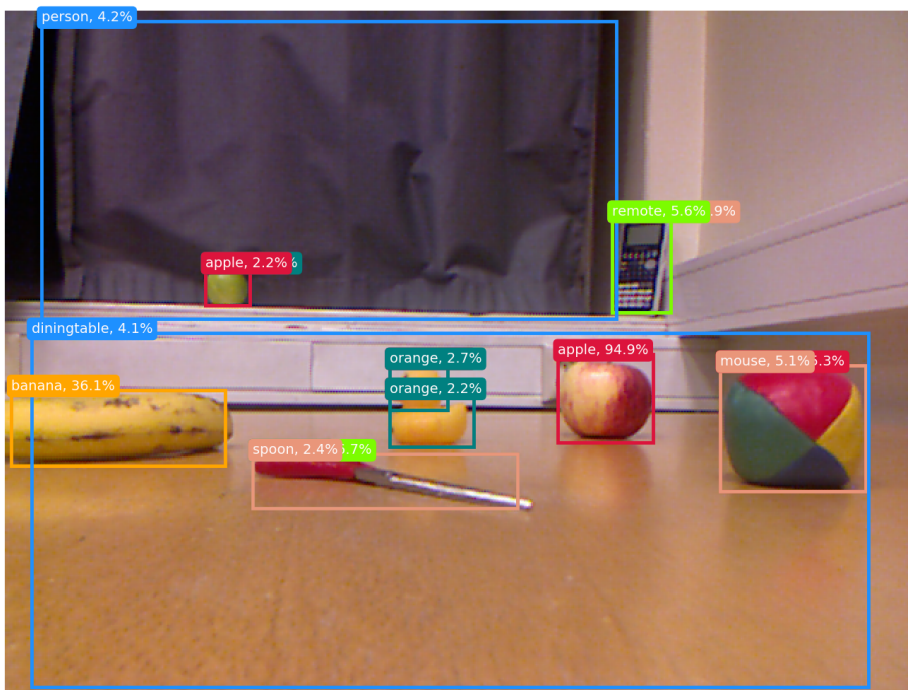
### 4.3.3 Object Detection

To use the Kinect in the perception module, it had to be able to detect objects of interest within its images. The task of object detection is a combination of localization and classification tasks: Objects of interest in an image should be localized, ie. marked within the image by a bounding box, and the localized objects should be correctly identified by choosing the correct class.

If the task of object detection is expected to run in real-time on a stream of images, detecting objects in images has to be a fast process. YOLOv3, presented in section 3.6, is a very fast object detection system, running at 30 FPS [36]. Using only a CPU, the analysis of a single image took 12 seconds. Real-time object detection requires

utilizing one or more dedicated GPUs [36]. It has been shown to perform at the level of other state of the art methods, but being very fast makes it suitable for real-time usage.

YOLOv3 has been trained on many different data sets, one of them being the COCO data set [25]. In the COCO data set, there are 80 classes:  $C = 80$ . Setting  $B = 3$ , each cell in the YOLOv3 system at every detection layer predicts  $5B + C = 255$  values. In order to limit the amount of predictions, a threshold of the certainty of the predictions can be set. By choosing a high threshold, fewer objects will be predicted, but the predictions are more certain. Using these settings, YOLOv3 was used on the RGB images captured by the Kinect.



**Figure 4.9:** An example of object detection of many objects. The threshold was set to 1% so that a wide range of predictions are allowed. The only predictions with high certainty are of the banana and the apple.

Between the 80 classes of the COCO data set, recognition by YOLOv3 is of varying quality. To find easily recognizable objects, tests were performed in which several



		Distance from camera			
		50 cm	75 cm	100 cm	125 cm
<b>Apple</b>	100% visible	99.4	96.6	96.9	33.8
	75% visible	97.4	37.5	41.0	-
	50% visible	76.5	6.4	15.9	-
<b>Banana</b>	100% visible	97.3	83.2	35.3	-
	75% visible	78.7	80.2	5.4	-
	50% visible	81.3	72.4	-	-

**Table 4.1:** The performance of YOLOv3 was tested with apples and bananas at different lengths away from the camera, and being partially visible when covered by a piece of paper. The results are given as numbers between 0 and 100, representing the confidence level of the correct prediction: 100% would correspond to the neural network being 100% sure of the correct prediction. "-" represents either a wrong prediction or no detection at all.

objects were laid out in front of the RGB camera to be recognized. The surroundings of the objects were similar to what can be found in a normal room: A table, walls and window curtains. An example of such a test is seen in Fig. 4.9. Apples and bananas were found to be the most accurately predicted object classes, and were therefore used in the project when testing the perception module.

### Object Detection Performance

In order to be familiar with the performance of YOLOv3, and to tune the threshold correctly, a series of tests were designed. Using the perception module to identify objects, it was important that the detection of the objects is stable, ie. that the objects would be accurately recognized at different locations in the room and while being only partially visible.

Performing several tests of fruit at different lengths away from the Kinect, as well as partially covering the fruit to different degrees, the limits of the object detection were explored. The results are presented in Table 4.1.

From Table 4.1 it is seen that the most reliable predictions were found for objects between 50cm and 75cm mostly also when being partially visible. For objects further away than 1m from the camera, detection was very unreliable. The tests of the system would therefore be performed at distances near 50cm, at most 1m away from the



**Figure 4.10:** Apples and bananas were identified at different lengths away from the camera and being partially covered. The threshold was set to 5% in order to only reduce the most unlikely predictions. It is likely that the network mistakenly identified the curtains as a jacket or a pair of trousers, therefore labeling it a person. Another mistake was the prediction of an orange, that was in fact a partially covered apple.

Kinect.

#### 4.3.4 Calibration of the Kinect

The intrinsic camera parameters, given by (2.14), of the RGB and depth cameras are different. The difference is visible by carefully comparing Fig. 4.7a and Fig. 4.7b. The RGB camera has a wider lens than the depth camera, ie. the focal length  $f$  is smaller. Moreover, the two sensors are located 2.5cm apart. This means that mapping pixels from one of the image planes onto the other requires knowledge of the intrinsic param-

eters of both cameras as well as the translation matrix between the camera centers.

An advantage of using the Kinect and its large and well-known community-driven toolkit Libfreenect, is that estimations of these intrinsic parameters and translation matrices have already been found and implemented in Libfreenect. Mapping every pixel from the RGB image onto the depth image plane, the result can be seen in Fig. 4.11.



**Figure 4.11:** The RGB image taken by the Kinect was mapped onto to its depth sensor plane. For pixels in which the depth sensor had no information, the color is left black.

Because the accuracy of the perception module was of high priority, manually calibrating the Kinect was considered. The accuracy gained by calibrating the Kinect was investigated by Branco Karan in 2015 [21]. It was found that manually calibration of the cameras greatly improved depth accuracy at distances far away from the Kinect. However, at small distances Karan found the error to be small: At a distance of about 0.97m, the uncalibrated Kinect had an error in depth prediction of 1.7cm. For the research in this thesis, the distance between the Kinect and the objects to be located was usually not more than 1m, and manual calibration was therefore not deemed necessary.

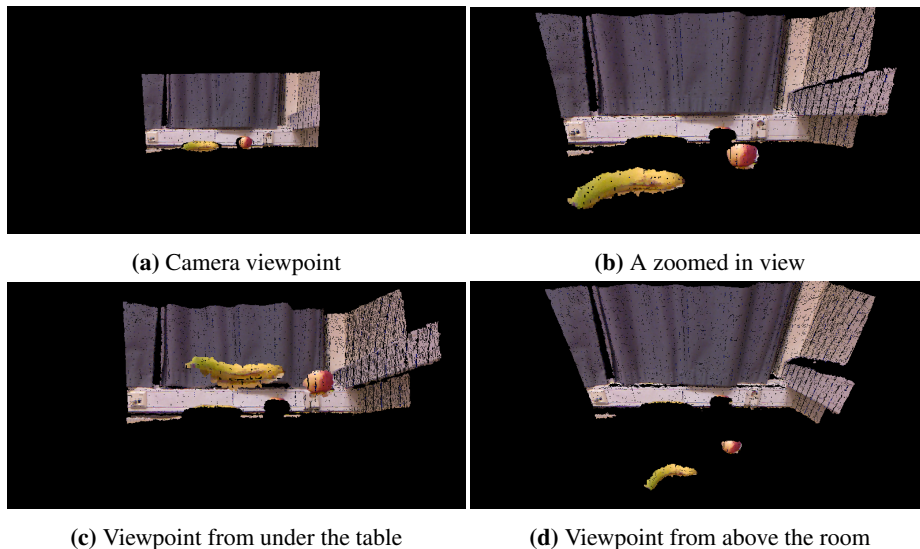
### 4.3.5 3D Point Localization

The coordinates of a pixel on the depth sensor plane is described by  $(x_d, y_d)$ , in which  $(0, 0)$  is the center of the depth image. At every pixel location, the depth sensor records

depth information  $z(x_d, y_d)$ . The projection from the three-dimensional room onto the depth camera plane is given by (2.12). To find the 3D location of the points projected onto the 3D plane, the task is simply to find  $\vec{x}_{3D}$ , given by (2.11). The formula for this projection is simple:

$$\vec{x}_{3D}(x_d, y_d) = \begin{bmatrix} x_d \cdot z(x_d, y_d) / f \\ y_d \cdot z(x_d, y_d) / f \\ z(x_d, y_d) \end{bmatrix} \quad (4.6)$$

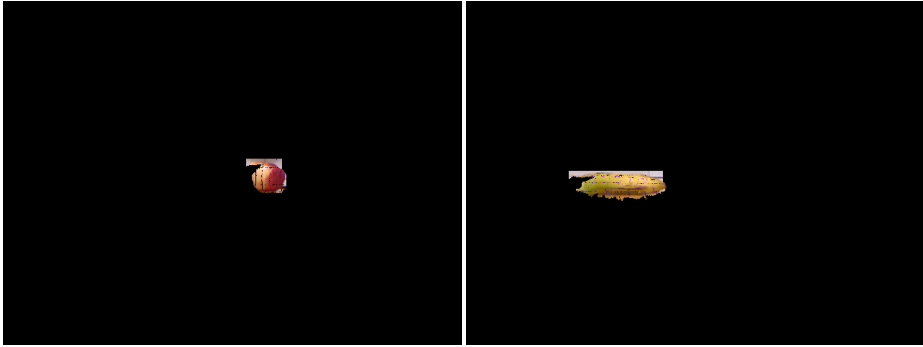
Projecting every pixel of Fig. 4.11 to three-dimensional space, a point cloud of the room was created. Using the Pyntcloud toolbox, this point cloud was visualized. The visualization of the point cloud created from Fig. 4.11 is seen in Fig. 4.12.



**Figure 4.12:** The point cloud created by projecting points from the Kinect’s depth camera is visualized.

It is not efficient to generate the full room seen by the depth sensor in order to locate objects in 3D. Instead, YOLOv3 was run on the initial RGB image taken by the Kinect camera to find the coordinates of pixels that contain interesting objects. The pixel locations of the detected objects were thereafter mapped over to the depth image plane, one object at a time. The result of this mapping is seen in Fig. 4.13, in which the

apple and banana have been localized separately on the depth image plane.



(a) An apple on the depth image plane

(b) A banana on the depth image plane

**Figure 4.13:** An apple and a banana have been separately mapped onto the depth image plane.

Two methods were used to determine the location of the objects in 3D. The first method was to find the average 3D position of the pixels making up the object, ie. the non-black pixels in Fig. 4.13. The other method sorted the  $x$ ,  $y$  and  $z$  location of every non-black pixel separately, then choosing the median value in every direction to represent the position of the whole object. The results of tests using the two methods are presented in Section 5.1.2.

## 4.4 The Planning Module

The planning module brings purpose to the actions of the robot: Its task is to automatically create a series of actions that the robot can perform in order to bring the environment to a desired goal state. In order for automatic planning to take place, the domain description and the problem description, as specified by the PDDL standard, have to be prepared. Parts of this was done manually by the user of the system, whereas other parts were done automatically by the planning module. This section will describe the implementation of the domain and problem descriptions used in this thesis.

### 4.4.1 Manual Domain and Goal Specification

The first step of the planning problem design is a general description of what the world looks like. This is what belongs in the domain specification: The robot itself and all possible types of objects that the robot could interact with, what states they could be in, as well as what actions the robot could perform on each object type, was specified in the domain description. This was done manually by the user of the system, as learning all possible states of the objects as well as finding the available actions at each state automatically for the robot would be a highly complex task.

Specifying the goal state was also done manually. This is natural, as the robot system was designed to perform tasks specified by a human. However, a script for automatic goal specification could be written. For example, a goal state could be to collect all apples into one basket and bananas into another basket. Removing or adding apples or bananas during the execution of the plan would then update the goal state to be suited to the new number of fruit on the workspace.

### 4.4.2 Automatic State Estimation

Receiving input from the perception module, the planning module could automatically estimate the initial state of the problem description. The input from the perception module contained information on all visible objects, as well as their location in the workplace. Estimation the state was done by comparing the location of objects relative to each other to each predicate. For example, an object *A* located above and sufficiently close to an object *B* would make the predicate "A on top of B" true. The full set of predicates made up the full state description, which was fed to the planner as the initial state of the problem.

### 4.4.3 Lookahead Planning

In order to be used in a real-world setting, lookahead planning as introduced in Section 2.4.6 was implemented into the planning module. Applying lookahead planning, the planner would be fed the observed state of the workspace after every finished action. This allowed for implementation of execution monitoring. If the estimated state based on this camera input differed from the expected state after performing an action, replanning could be performed. When the robot needed to replan, finding the new initial state could be done automatically by the automatic state estimation, leaving no need for human intervention after starting the program.





# Chapter 5

## Testing

### 5.1 Module Tests

#### 5.1.1 Robot Accuracy and Operational Challenges

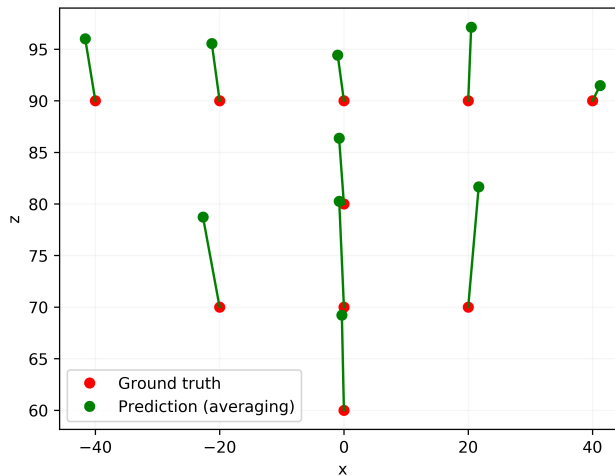
This section was supposed to contain measurements of the accuracy of the control module, measured by the following method: The robot would be placed on top of graph paper aligned with the robot Cartesian coordinate system. Marking points on the graph paper with a pen, the robot would be instructed to move to this point in the robot Cartesian coordinate system. The accuracy of this test would be an indicator of how well the inverse kinematic calculations and PID controllers were working.

Unfortunately an accident occurred, after which the robot fully stopped communicating with the computer. The accident happened because of an attempt to correct an interference error that occurred when connecting both the Kinect and the robot to the computer at the same time. The interference blocked a large portion of the messages sent to the robot, and so the system was very slow. It was suggested that the interference was caused because both the connection between the computer and the robot as well as the connection between the computer and the Kinect carried a current of 5 volt as well as ground. A special connection between the robot removing the wire leading current between the computer and the robot was created at the laboratory, and after experimenting with it for a while, although seemingly working at first, the robot

stopped responding after a while. It is believed that the custom-made setup overheated the motors or led to high voltages being sent to the actuators.

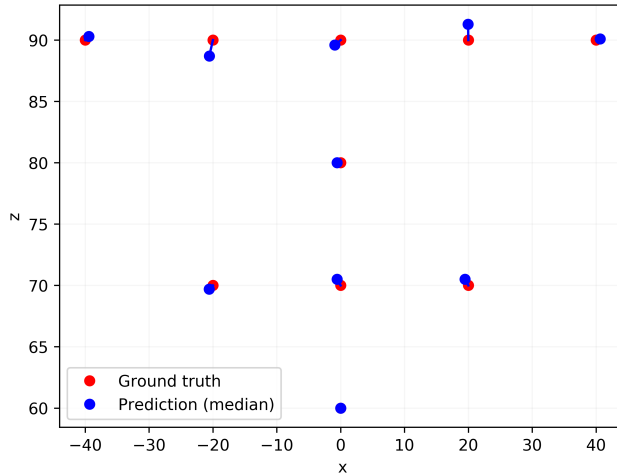
Although the robot accuracy was not measured in detail, the robot was made to perform some easy tasks before it broke down. It was able to move fruit from one location to another by picking it up, holding it while moving and then releasing it when arriving at the goal location. However, when gripping apples even a small displacement from the mass center of the apple would cause it to roll out of the grippers. This could possibly be fixed with a modification of the grippers, either by making them wider or by adding cushions to the tips to improve their grip.

### 5.1.2 3D Localization Accuracy



**Figure 5.1:** 3D localization using the mean of the pixels belonging to the object as found by the object detection algorithm. The axes are showing the Kinect coordinate system, one unit corresponding to 1cm. The line between the ground truth and the predicted location of the apple indicates the error in the prediction. It is seen that the error is generally large, although highly varying.

The perception module was tested by the accuracy of its 3D localization of objects. The Kinect was placed on top of graph paper aligned with the Kinect's coordinate system. Apples were placed on certain locations on the graph paper, and the perception



**Figure 5.2:** 3D localization using the mean of the pixels belonging to the object as found by the object detection algorithm. The axes are showing the Kinect coordinate system, one unit corresponding to 1cm. The error between the ground truth and predicted location is shown to be small, at most deviating by 2cm from the ground truth.

module was run to predict the location of the apples. As described in Section 4.3.5, this prediction used either an average of the location of every pixel belonging to the object, or the median of the pixel locations. The result using the mean method is seen in Fig. 5.1, and the results using the median is seen in Fig. 5.2.

As seen in Fig. 5.1, the average method yielded unreliable, highly varying results. Especially the location along the  $z$  axis is overestimated at every location. The cause of this is likely that the object detection algorithm does not perfectly isolate pixels belonging to the object, but rather cuts out a square surrounding the object, therefore also including pixels from the background. In the context of the test in Fig. 5.1, pixels belonging to the wall seen behind the apples are located far behind the pixel belonging to the apples, thus influencing the average location greatly and causing the error.

The median method in Fig. 5.2 circumvented the problem of extreme outliers, showing that the depth sensor in the Kinect indeed provides accurate depth information.

### 5.1.3 Planning Tests

A planning domain that could be used to test the performance of the planning algorithms was designed. The design was inspired by the blocks world automated planning problem, in which blocks on a table can be picked up and stacked on top of each other in order to build specific stacks of blocks.

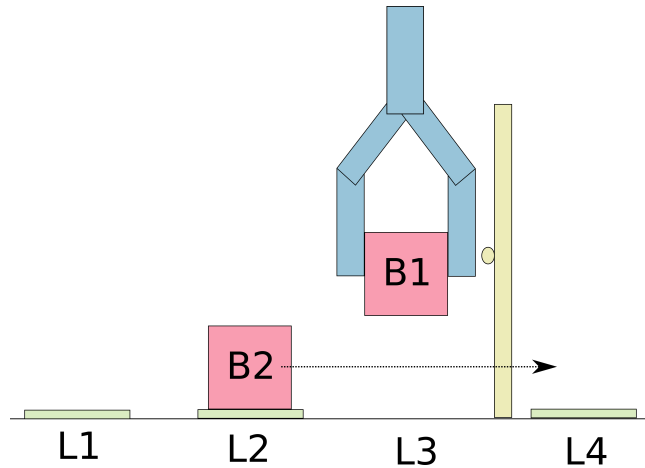
The domain contains the robotic manipulator as well as the following objects:

- **Locations:** Locations are necessary to specify where objects are located. A box on the ground is at any time in a given location, and a door leads from one location to another. To provide spatial information, all adjacent locations are specified in pairs.
- **Boxes:** Boxes are objects that can be picked up and moved around by the manipulator. They are either situated at a certain location or being held by the manipulator grippers. They can be alone on the ground or stacked on top of each other.
- **Doors:** Doors are located in between two adjacent locations. Whether the manipulator can move between these locations depends on whether the door is open or closed.

The actions describe the possibilities of what the manipulator can do, and are as follows:

- **Move:** The manipulator can move between adjacent locations.
- **Pick up box:** The manipulator can pick up a box if there is a box at the same location. If several boxes are stacked on top of each other, the manipulator can only remove the top one.
- **Put down box:** If the manipulator is holding a box, it can drop it off at the current location. If there are already boxes at the location, the released box will be on the top of the stack.
- **Open or close door:** If the manipulator's grippers are empty, ie. it is not holding any box, it can open a door that is next to its current location.

After specifying the domain, the planning problems could be constructed. One example of a simple planning problem is shown in Fig. 5.3. In this problem, the robot



**Figure 5.3:** The initial state of a simple planning problem. The goal state is indicated by the arrow: A goal state is a configuration in which block *B2* is located at *L4*.

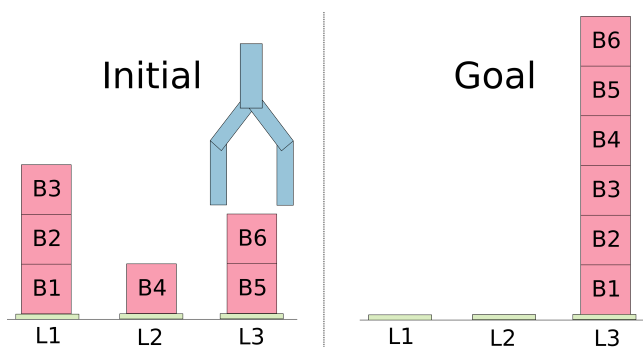
needs to put down the block it is holding, open the door and then bring the other block to the final location. Running the planner using a number of different heuristics and search algorithms gave the results seen in Table 5.1.

Performance on a simple planning problem				
Search algorithm	Heuristic	Nodes expanded	Search time	Plan length
BFS	-	37	0.0018 s	11 steps
A*	hMax	21	0.023 s	11 steps
A*	hAdd	22	0.021 s	11 steps
A*	FF	19	0.021 s	11 steps
GBF	hMax	18	0.020 s	13 steps
GBF	hAdd	16	0.016 s	15 steps
GBF	FF	17	0.019 s	15 steps

**Table 5.1:** BFS, A\* and GBF search algorithms using different heuristics were evaluated on the simple planning problem in Fig. 5.3.

Although the problem is simple, the algorithms show their characteristics in the results of Table 5.1. The problem is very small, and so the blind breadth-first search is

significantly faster than the other algorithms even though it expands the largest number of nodes. This is because calculating the heuristics is expensive in every other algorithm. As described in Section 2.4.4, BFS will always find the optimal solution, ie. the plan with the lowest number of steps. It is apparent that the A\* algorithm is careful in choosing which nodes to expand and therefore finds the optimal solution for this problem. GBF does not find the optimal solution using any of the heuristics, however it expands the fewest nodes. There is only one optimal solution to the simple problem of Fig. 5.3: The robot should first put down block *B1* at location *L1*, open the door, and finally move block *B2* to location *L4*. GBF using the FF heuristic, however, found a solution in which block *B1* was placed on top of *B2* before opening the door, thus requiring extra steps for removing *B1* afterwards.



**Figure 5.4:** The initial state and the goal state of a more challenging planning problem.

A larger planning problem in the same domain was designed as seen in Fig. 5.4. The optimal solution for this problem involves an intricate series of actions in order to end up with the correct stack of boxes at the end. The same search algorithms and heuristics as were used for solving the problem in Fig. 5.3 were applied to this problem, and the results are given in Table 5.2.

The only two algorithms that can guarantee to find the optimal solutions in Table 5.2 are BFS and A\* search using the admissible hMax heuristic. However, all of the A\* algorithms perform well, but although the A\* algorithm is precise, its run-time is far greater than that of the GBF algorithms. The FF heuristic led to the best performance among the GBF search heuristics.

Similar problems to the one in Fig. 5.4 were created, starting from a domain with only one block. Each problem added one more block to the domain. The time complexity of the search algorithms run on these problems is shown in Fig. 5.5a. The number

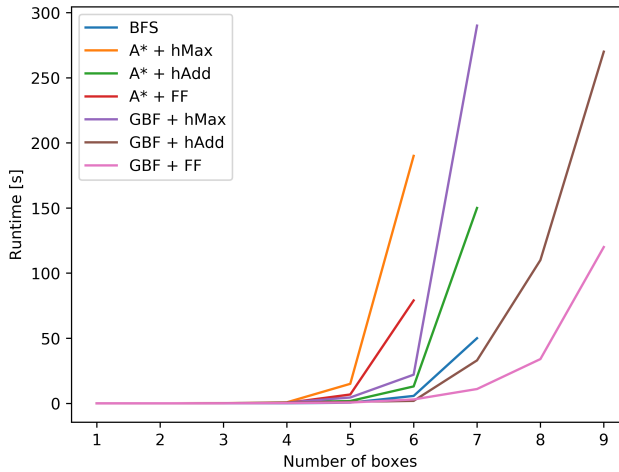
---

Performance on a difficult planning problem				
Search algorithm	Heuristic	Nodes expanded	Search time	Plan length
BFS	-	70485	5.7 s	50 steps
A*	hMax	40245	190 s	50 steps
A*	hAdd	2881	13 s	58 steps
A*	FF	17401	79 s	50 steps
GBF	hMax	4591	22 s	90 steps
GBF	hAdd	441	1.9 s	102 steps
GBF	FF	699	3.0 s	82 steps

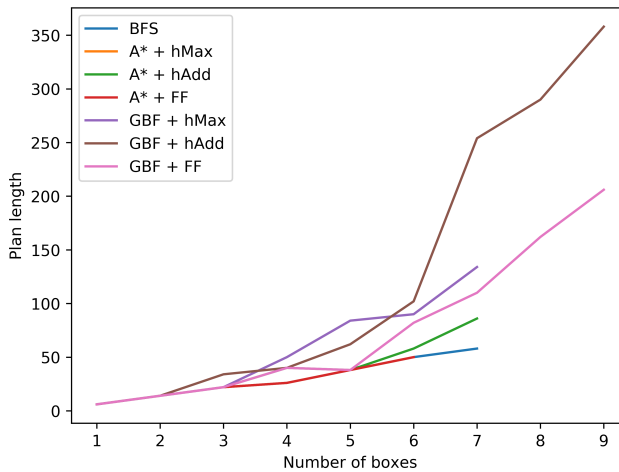
**Table 5.2:** BFS, A\* and GBF search algorithms using different heuristics were evaluated on the planning problem in Fig. 5.4.

of steps in the solution found by the search algorithms is presented in Fig. 5.5b.

The increase in run-time is highly exponential, which was to be expected by the rapid growth of the number of possible states each block adds. The BFS is the fastest algorithm, and with its optimality guarantee this makes it suitable for small domains. For larger domains with many possible states, the GBF search algorithm using the FF heuristic seems to perform well in addition to short time usage.



(a) Run-time of the planning algorithms. A limit of five minutes run-time was set, resulting in most algorithms only being able to solve the problems up until 6 or 7 boxes were included. Only the GBF algorithm was able to find a plan for the domain using 9 boxes.



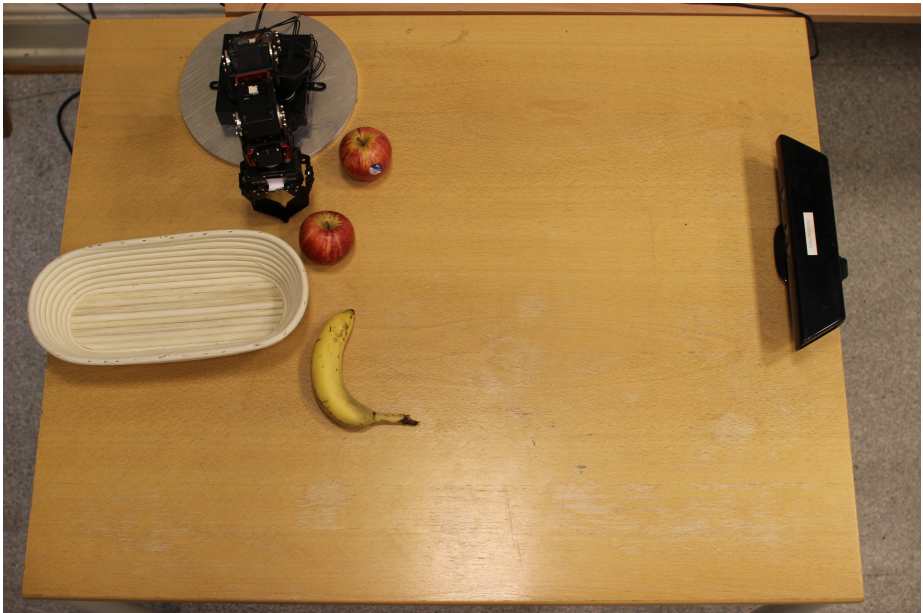
(b) The length of the plans created by the algorithms. The A\* search algorithms generally created better plans than the GBF algorithms.

**Figure 5.5:** Run-time and length of the plans created by various planning algorithms on a number of different problems.



## 5.2 Case Study

The set-up, execution and results of a case study, **stacking fruits in a basket**, will be discussed in detail in this section, in order to evaluate the full system performance on a real-world problem.



**Figure 5.6:** The setup of the fruit stacking problem. The camera and the robot are located in known locations so that the transformation matrix between the two coordinate systems can be found.

### 5.2.1 Problem Specification

Starting with a number of apples, bananas and a basket on a table, the task of the problem is to move every fruit into the basket. However, as bananas take up much space in the basket if not they are lying diagonally, it is important that the bananas are put at the bottom of the basket so that they do not balance on top of apples. The goal of the fruit stacking problem is therefore to insert all fruits into the basket so that all bananas end up at the bottom at the basket with the apples on top.

A number of assumption were made in order to make the problem solvable for the designed robotic manipulator system:

- There is a safe zone above the work space, from 30cm above the table and above, in which the robot can move freely without risking to run into any objects.
- The fruit bowl acts like a stack - a LIFO structure. The last fruit added into the basket must also be the first one to be removed.
- No fruits are stacked on top of each other on the table, and every fruit can be grabbed and released by firstly lowering the grippers from a point directly above the fruit.
- All fruits on the table can be seen by the camera at some point during the plan execution. An apple blocking the view of a banana is fine as long as the banana will be visible after moving the apple.
- The contents of the basket cannot be seen by the camera. Therefore, adding or removing fruits into the basket cannot be seen by the camera, and this can therefore not be handled. Moving, adding or removing fruits on the table, however, can be handled by the system. Only manipulation of the fruits on the table was therefore allowed.

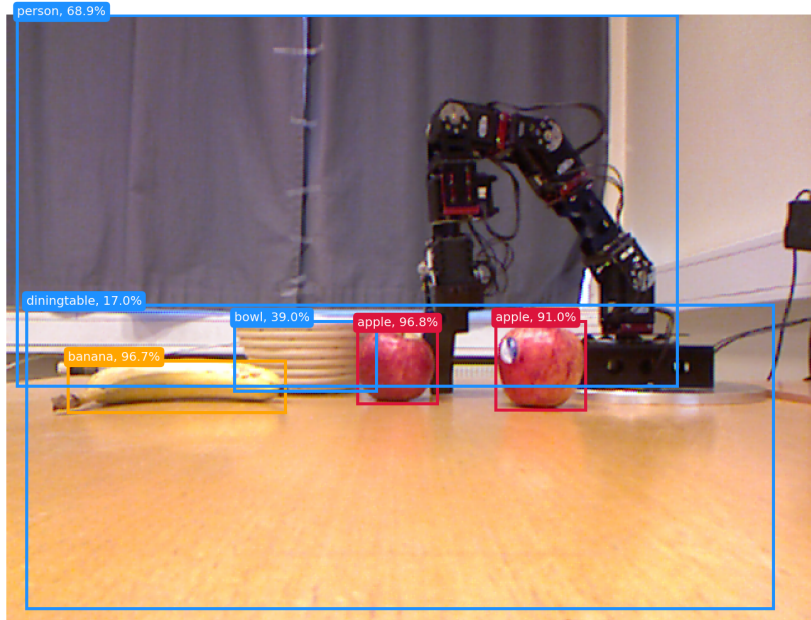
The setup of the robot and camera, along with a fruit bowl and several fruits on the table, can be seen in Fig. 5.6.

Because the robotic manipulator was unavailable at the time, the actuator communication in the control module was disabled, and instead of letting the robot move the items, they were moved manually by hand. This allowed for testing of the whole system except for the control of the manipulator.

### 5.2.2 Solution

In order to create a flexible solution to the problem of stacking fruits in the basket, a variant of Run-Lookahead, Algorithm 2.2, was implemented.

Firstly, the domain of the problem was specified for the planning module. Only two object types in the environment were modeled: *Fruit* and *fruit bowl*. This way, adding new types of fruits would be simple as all actions associated with any kind of fruit such as *gripping*, *releasing*, *holding* and so on are similar for all fruits. Any special

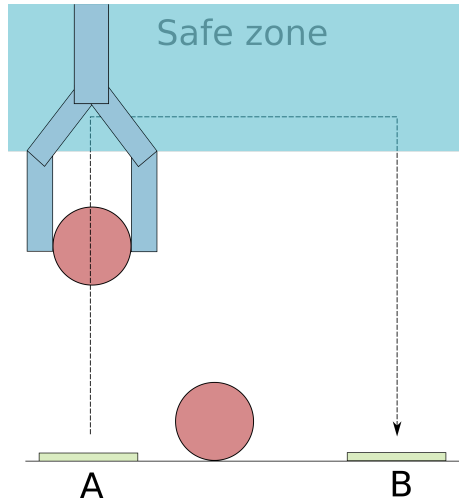


**Figure 5.7:** Object detection through the lens of the Kinect, its threshold set to 15%. In addition to detecting the fruit and the basket on the table, other detections were occasionally made. Although those detections were discarded in this experiment, they could have been kept and used to give information on unexpected objects in the workspace.

technique for handling a certain fruit type, such as performing a tighter grip for apples than for bananas, could easily be added separately into the actor module.

The perception module identified and located objects on the table. The scene shown in Fig. 5.6 seen through the lens of the Kinect is shown in Fig. 5.7. The object types not related to *fruit* or *fruit bowl* were discarded, and the location of each remaining object was stored. Using the location of each object found by the perception module, the scene was decoded and registered as the initial state in the problem description by the planner. The location of each object was not designed as part of the planning problem, but was rather stored as local variables by the planner. This way, changing the location of an object would not require replanning of the whole scene, but rather just relocating

that specific object and continue the plan.

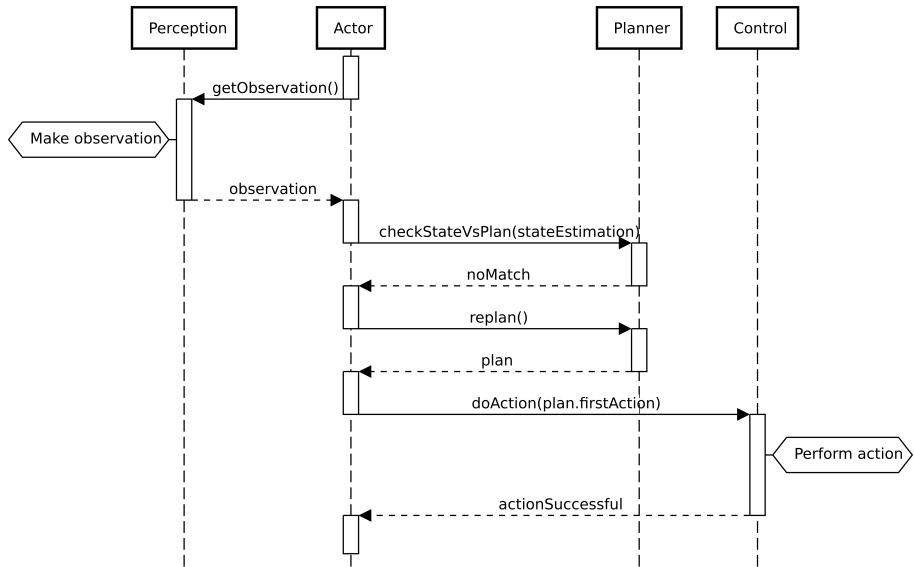


**Figure 5.8:** Under the assumption that there are no obstructions from 30cm and upwards above the table, the robot was programmed to move in a "square" pattern in order to avoid collisions.

Running the plan yielded a sequence of actions, if possible, such that the fruit would be correctly stacked in the basket with the bananas at the bottom. The actor converted the first action of this sequence into corresponding orders to the control module. To simplify control of the robot while at the same time hinder that the robot crashed into objects while moving, the movement of the robotic arm was restricted as follows: When handling objects on the table or in the basket, the robotic arm would be lowered straight down from the safe zone and handle the object from above, the grippers facing down towards the table. Moving from one location on the table to another would therefore be performed in a "squared" path. This path is visualized in Fig. 5.8.

Execution monitoring was performed. After each action, a new scene image analyzed by the perception module was compared to the plan created by the planner. Because the contents of the basket could not be seen, the image analysis was therefore firstly augmented to include the contents of the basket. If the perceived scene and the expected scene matched, the next action of the plan could be performed. However, if objects were found to be removed or added on the table, replanning was initiated.

A sequence diagram of the control flow when performing a single action of a plan is visualized in Fig. 5.9. In the example in the diagram, the observed state by the camera



**Figure 5.9:** A sequence diagram of the control flow before and during the execution of a single step in the plan.

differed from the expected state from the previously created plan, and therefore the planning module had to replan. If the expected state matched the observed state, the next action could simply have been performed without replanning.

A robot working in a real-world environment has time constraints. Therefore, the processing of information while the robot is waiting for new commands needs to be fast. An analysis of the time spent for every step in the example modeled in Fig. 5.9 was performed. The results are shown in Table 5.3. It is seen that object detection was a slow process. However, as described in Section 4.3.3, using a GPU would bring the processing time down to a fraction of a second. 3D localization was also found to be a slow process. Improvements that can be made to the 3D localization methods described in Section 4.3.5 might optimize the speed significantly. These improvements are mentioned in Section 6.3. Lastly, the fastest step in the pipeline is the replanning. However, it is worth noting that its runtime is highly variant on the number of objects present, and that the time taken to replan increased when many objects were present, as was seen in Fig. 5.5a. For this experiment, only three fruits and the basket were present.

Performance	
Subtask	Time elapsed [s]
Object detection	11.54
3D localization	17.38
State check and replan	0.098
<b>Total time</b>	<b>29.02</b>

**Table 5.3:** The example modeled in Fig. 5.9 was simulated. Object detection and 3D localization took a long time to process, whereas the replanning was very fast.

### 5.2.3 Handling Complex Situations

The system was designed to be able to handle a variety of scenarios that might occur in a real-world environment. One example of dealing with unexpected behavior happened when one or several of the items on the table were moved from their original position to a new position on the table. Because the position of the objects were abstracted away from the plan itself and rather stored internally by the planning module, the planning module would in this case just update the location to which the rearranged objects were pointing without the need of potentially expensive replanning.

Regular control on whether the environment has reached the expected state after every action of the plan has been carried out also helped handling unexpected events. This way, instead of blindly carrying out the full plan as soon as it was been created, the system took one step at a time, making sure that any visibly unforeseen events were captured and taken into account. For example, removing or adding new fruits onto the table required the system to figure out a new plan on how to place the fruits into the basket, thus requiring to replan. A situation in which adding a single fruit has a big impact on the plan, is when the basket is already full of apples before a banana is added on the table. Replanning would then find a plan in which every apple was to be taken out of the bowl before the banana could be inserted, before finally putting the apples back into the basket. A similar situation that is handled by the system is a fruit blocking the view of other fruits. Once the visible fruit has been added into the bowl, the now visible fruit will trigger a replan, thus handling the situation in the end.

### 5.2.4 Limitations

A significant limitation of the system test is that the control module could not be tested. Although the control module was able to move fruits between different locations on the table, adding fruits into the basket and removing it from the basket is a more difficult task that might require small changes to the control module.

The processing time between every action, as shown in Table 5.3, is significant. Using a GPU, object detection could be performed in a fraction of a second. However, finding the 3D location of the objects would still have a processing time far higher than desirable for a robot operating in a real-world environment, reacting to events in real-time.

The system is highly reliant on the perception module to detect every object in the workspace. However, when there were many objects in an image, partially blocking each other, detection became unreliable. As was seen in Table 4.1, partially blocking the objects led to a significant drop in detection confidence. Moreover, object detection was sensitive to the lighting conditions, failing to detect objects in strong sunlight or darkness.

Suggestions on improvements on these issues are given in Section 6.3.





# Chapter 6

## Discussion

Detailed discussion regarding implementation was given in Chapter 4. This chapter will discuss the main results in light of the project's goals in section 1.3 and present suggestions on future research.

### 6.1 Goals and Test Results

In section Section 1.3, three project goals were specified: A study into fundamental theory and recent research on AI planning and robotic control, setup and implementation of the robot, camera and planning software, as well as merging of the implementation into a full working robot system.

The first goal was to gain an understanding of the fields of AI planning, artificial neural networks and robotic control. During the fall semester of 2017, deep AI neural nets were explored, and having a background in engineering cybernetics, many aspects on robotics control have been covered in earlier courses. These sources were revisited and studied. However, having no previous experience with AI planning, an online course by the University of Edinburgh [47] was followed in order to understand fundamentals of the field. Recent research was studied in order to understand how these fields could be combined in meaningful ways. Chapter 2 can be seen as a summary of the theory studied during the course of the project.

The second goal was to set up the relevant hardware and software tools used for the project. As the thesis was largely a practical project, a significant amount of time was spent on issues related to the hardware: Setting up the hardware and researching how to use Python and software tools for communication with the actuators and camera was a tedious task at times. Moreover, a significant portion of the time was spent debugging issues with hardware communication. Setting up the hardware from scratch was a more difficult task than expected, limiting time available for creating more complex software. Creating a computer simulation of the whole system instead of working with hardware would free up much of this time, allowing time to improve on each module. However, as the objective was to merge high-level AI with low-level control, working in a real environment and making all parts work together was more rewarding.

Finally, the goal of creating a fully autonomous robot system was attempted. After working with the modules separately for a long time to set them up, making the modules communicate and interact as a single system was a success for achieving the thesis' objective. The perception module and the planner worked as intended, continuously evaluating the environment through images, comparing them to the expected state of the world and replanning if unexpected events had occurred. Although the robot was unavailable during the case study, its software has been prepared, and so including it into the full system should be an easy task.

## 6.2 Design

The design of the robot system was separated into the three modules of control, perception and planning. By separating the system into modules and letting each module handle specific tasks, the system is scalable. Fixes or extended functionality can easily be done by just altering a single module. For instance, if one wants to use another camera than the Kinect or another robot than the custom built robot used in this thesis, it is simple to change functionality in the relevant modules. Moreover, it is possible to add new modules to the system. This way, the design used in this thesis can be used as a basis for more complex designs.

There are additional advantages to the modular design, including error isolation, readable code and reusability of modules. In fact, the error isolation was important to the testing of the project: When the robot broke down and was rendered unusable, the error was confined within the control module, allowing testing of the other modules without changing the implementation. Moreover, the system will work with any type of robot manipulator. As the information about the size and build of the robot was

stored inside the URDF file of the control module, using another robot would simply just require a new URDF file describing the new robot, as well as an updated function for reading this file inside of the control module.

## **6.3 Future Work**

As the project was large with many components and modules to combine, working with a time constraint did not leave enough time to optimize the performance of the robot, and each of the modules is believed to have the potential to be better optimized and to be adjusted to allow for more high-level and complex behavior. This section will present ideas for improvement and further development of the system modules. Some of the suggestions are made to fix the limitations of the system pointed out in Section 5.2.4. Also, suggestions for new functionality will be presented.

### **6.3.1 Control Module**

Although the control module was not tested thoroughly, it was noted that the grippers struggled with holding without losing them. Augmenting the physical grippers by attaching cushions or sticky material to the grippers could be a solution to this, although it might be possible to fix the issue through software alone. For instance, it would be interesting to investigate whether AI methods could be used to optimize the gripper technique depending on the objects.

### **6.3.2 Perception Module**

As the object detection of the perception module could not recognize the robot grippers, controlling the robot was performed open-loop: The camera location and the base of the robot had to be placed at an exact distance with a specific rotation of their coordinate systems. Any error in the camera model, robot model or the model of the displacement and rotation of their coordinate systems would lead to errors in control. However, retraining the YOLOv3 network to recognize the robotic arms could provide closed-loop control of the robot: The camera could simply calculate the distance between any objects the grippers are moving towards, letting the robot control for this relative distance.

Described in Section 5.2.4, a limitation of the perception module is its inability to

detect all objects in bad lighting conditions or with many objects present. Training the object detection neural network on training data images taken in similar conditions can improve the performance. Another solution would be to mount a camera on the robot itself, allowing it to move around in order to change the camera's viewpoint. Also, using several cooperating cameras located at different locations could be an easy way to improve the detection performance.

It was mentioned in Section 5.2.2 that 3D localization was performed inefficiently. Two suggestions for speeding up the processing time are presented: Firstly, an improvement to the current implementation would be to map all detected objects in the image simultaneously onto the depth plane and then into 3D space. Currently, only one object is handled at a time, increasing the processing time depending on the number of objects in the image. Secondly, the projection of objects into 3D used an input of the full image size even if most of the information was empty, visualized as black pixels in Fig. 4.13. The implementation could greatly improve by only using the necessary pixels for these calculations.

### 6.3.3 Planning Module

Also the planning module could be modified to handle more complex situations. For example, the control module might report after several tries that an action is hard to perform or takes a long time. An intelligent planner could then weight plans depending on whether they include that specific action, thus learning to avoid performing difficult actions after a while.

For complex environments, creating plans might take a long time. Therefore, replanning is also expensive. Sometimes, events occur in the environment that can be easily fixed, such as an obstruction appearing and blocking the current plan execution. In this situation, the replanning can be designed so that its goal is to bring the environment back to its previous state and from there continue executing the original plan, for example by removing the obstruction. This form of replanning design can be a great time saver, and might be worth researching.

## 6.3.4 Additional Functionality

### Modules running on separate threads

The system would be suited for a design in which each module was run as a separate thread, using message-passing for communication between the modules. This would allow the perception module to be constantly updated, even while the robot is moving. Moreover, instead of an actor having to ask the perception module about the current situation, the perception module could rather be responsible itself for notifying unexpected observations, saving time for the actor. ROS, the Robot Operating System, is a framework allowing for this design [34].

### Path Planning

The inclusion of a path planner could be an interesting addition to the system. Instead of relying on a safe zone above the workspace, a path planner would rely on the perception module to detect obstructions between the start point and the end point, allowing the gripper to move shorter distances than in the current system.

### Reinforcement Learning

Having worked mainly with reinforcement learning (RL) in my specialization topic during the fall semester of 2017 [15], I believe that there is great potential for RL techniques to be integrated into the system. Much like humans and animals learn by repeatedly perform an action over and over in order to perfect it, reinforcement learning allows learning through repetition of actions.

The following is a list of interesting possibilities for RL to improve the system:

- **Learning to control the gripper:** Reinforcement learning could be used for the grippers to learn optimal gripping power or the best angle for gripping an object.
- **Performing difficult control tasks:** The robot could use RL techniques to learn difficult control tasks. An example of such a task is leaning how to throw objects onto a target, giving a score depending on how close the objects are to hit the target.
- **Blending RL and AI planning:** RL and AI planning can be blended in a number of ways. For example, RL methods could be used on the planning graph

itself, improving the plan according to which actions were fast and safe and filtering out bad actions. Another idea is using RL to learn the actions given in the plan created by the planner, as was demonstrated in experiments by Ground and Kudenko [14].

It is worth noting that RL methods generally require a significant amount of training, and that repeating actions over and over for a physical robot takes a long time. Care should be put into the designs of the algorithms to reduce the required amount of training.

# Chapter 7

## Conclusion

In this project, an investigation of the fusion of robotics and AI planning has been carried out, firstly through studies of fundamental theory and research, thereafter by implementation of a controller, a perception module and a planner, and finally by a merging of the components into a robot system.

The study of theory and modern research guided the system design and choice of software tools used in the project, and it hoped that the theory presented in Chapter 2 will be a concise and valuable source for understanding the implementation of the project and the possibilities for expansions in future research.

The separate hardware components of the system were set up and enhanced by software tools allowing for high-level task performance. The robot could be controlled by the use of inverse kinematics, the perception module, using a Kinect camera, was used to detect objects and find their position in the environment, and the planner was able to automatically create plans of actions and replan when necessary.

After combining the components into a complete system, a test case was designed to evaluate the full system. Although the robot itself was unable to move during the final testing, the execution monitoring of the system was fully automatic and proved to be able to handle a range of complex situations that can occur in a real-world environment. Because of the modular design, the system was general and could be applied to any robotic manipulator by just changing the part of the control module responsible for modelling the robot.

Through future research, it is hoped that the modules making up the system will be continuously improved upon, and that additional functionality will be added to the system. The following is a summary of the suggestions for future work, presented in detail in Section 6.3:

- As the control module struggled with gripping objects, it would be interesting to investigate the possibility of using AI to help the robot to learn gripping techniques.
- The system could benefit by the perception module learning to identify the robot. This would allow for closed-loop control of the system.
- The AI planner could be enhanced by making it compare different plans based on prior experience. Also, its replanning functionality would be made more efficient by including the option to create plans leading back to a known state in an earlier, original plan.
- Using path-planning could help improve the efficiency of the robot movement while taking care of avoiding obstructions.
- Reinforcement learning techniques could be implemented in a number of ways, allowing the robot to learn through experience and self-improving when performing difficult control tasks or creating plans.



# Bibliography

- [1] Agatonovic-Kustrin, S., Beresford, R., 1999. Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research. *Journal of Pharmaceutical and Biomedical Analysis*.
- [2] Alkhazraji, Y., Frorath, M., Grützner, M., Liebetaut, T., Manuela, Ortlieb, Seipp, J., Springenberg, T., Stahl, P., Wülfing, J., 2010–2018. Pyperplan. [Online; accessed 04-February-2018].  
URL <https://bitbucket.org/malte/pyperplan/src/b014fe50dbbfe6949327f875e0d4a092f28fa72a/src/pyperplan.py?at=default&fileviewer=file-view-default>
- [3] Barrett, A., Weld, D. S., 1994. Partial-Order Planning: Evaluating Possible Efficiency Gains. *Artificial Intelligence* 67 (1), 71–112.
- [4] Behnel, S., Bradshaw, R., Dalcín, L., Florisson, M., Makarov, V., Seljebotn, D. S., 2018. Cython C-Extensions for Python. [Online; accessed 4-May-2018].  
URL <http://cython.org/>
- [5] Bengio, Y., 2009. Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning*.
- [6] Blum, A. L., Furst, M. L., 1997. Fast Planning Through Planning Graph Analysis. *Artificial intelligence* 90 (1-2), 281–300.
- [7] Bradski, G., Kaehler, A., 2008. Learning OpenCV. O'Reilly Media.

- 
- [8] Cybenko, G., 1989. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals, and Systems*.
- [9] de la Iglesia Castro, D., 2018. pyntcloud. GitHub repository, [Online; accessed 4-May-2018].  
URL <https://github.com/daavoo/pyntcloud>
- [10] Fikes, R. E., Nilsson, N. J., 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial intelligence* 2 (3-4), 189–208.
- [11] Fung, N., 2017. Toward Real Time Autonomous Robotics Through Integration of Hierarchical Goal Network Planning and Low Level Control Libraries.
- [12] Ghallab, M., Nau, D., Traverso, P., 2014. The actor’s view of automated planning and acting: A position paper. *Artificial Intelligence* 208, 1–17.
- [13] Ghallab, M., Nau, D., Traverso, P., 2016. *Automated Planning and Acting*. Cambridge University Press.
- [14] Grounds, M., Kudenko, D., 2008. Combining Reinforcement Learning with Symbolic Planning. *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning*.
- [15] Gulbrandsen, Ø. H., 2017. *Fundamental Deep Reinforcement Learning Algorithms*.
- [16] He, K., Sun, J., 2014. Convolutional Neural Networks at Constrained Time Cost. *arXiv preprint arXiv:1412.1710*.
- [17] He, K., Zhang, X., Ren, S., Sun, J., 2015. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385*.
- [18] Hoffmann, J., Nebel, B., 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14, 253–302.
- [19] Ingrand, F., Ghallab, M., 2014. Robotics and Artificial Intelligence: a Perspective on Deliberation Functions. *AI Communications* 27 (1), 63–80.
- [20] Itseez, 2018. Open source computer vision library. [Online; accessed 20-May-2018].  
URL <https://github.com/itseez/opencv>

- 
- [21] Karan, B., 2015. Calibration of Kinect-type RGB-D Sensors for Robotic Applications. *FME Transactions*, 43(1), pp.47-54.
- [22] Kim, B. S., 2017. About robotis. [Online; accessed 27-February-2018].  
URL [http://en.robotis.com/model/page.php?co\\_id=introduce](http://en.robotis.com/model/page.php?co_id=introduce)
- [23] Lallement, R., de Silva, L., Alam, R., 1994. HATP: An HTN Planner for Robotics. arXiv preprint arXiv:1405.5345.
- [24] Lekavý, M., Návrát, P., 2007. Expressivity of STRIPS-Like and HTN-Like Planning. *KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*.
- [25] Lin, T.-Y., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C. L., Dollár, P., 2015. Microsoft COCO: Common Objects in Context. arXiv preprint arXiv:1405.0312.
- [26] Manceron, P., 2017. IKPy. GitHub repository, [Online; accessed 4-May-2018].  
URL <https://github.com/Phylliade/ikpy>
- [27] Mankoff, K. D., Russo, T. A., 2013. The Kinect: a low-cost, high-resolution, short-range 3D camera. *Earth Surface Processes and Landforms*, 38, 926-936.
- [28] McDermott, D., 2000. The 1998 AI Planning Systems Competition. *AI Magazine Volume 21 Number 2*.
- [29] Nau, D. S., Ghallab, M., Traverso, P., 2015. Blended Planning and Acting: Preliminary Approach, Research Challenges. *AAAI Conference on Artificial Intelligence*.
- [30] Nielsen, M. A., 2015. *Neural Networks and Deep Learning*. Determination Press.
- [31] Nocedal, J., Wright, S. J., 2006. *Numerical Optimization*. Springer.
- [32] OpenKinect project, 2010–2017. libfreenect. GitHub repository, [Online; accessed 3-May-2018].  
URL <https://github.com/OpenKinect/libfreenect>
- [33] Puja, F., Grazioso, S., Tammara, A., Ntouskos, V., Sanzari, M., Pirri, F., 2017. Vision-based deep execution monitoring. arXiv preprint arXiv:1709.10507.

- 
- [34] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A., 2009. ROS: an open-source Robot Operating System. CRA workshop on open source software 3 (3.2), 5.
- [35] Rajan, K., Saffiotti, A., 2017. Towards a science of integrated AI and Robotics. Artificial Intelligence.
- [36] Redmon, J., 2013–2016. Darknet: Open Source Neural Networks in C. [Online; accessed 24-April-2018].  
URL <http://pjreddie.com/darknet/>
- [37] Redmon, J., Divvala, S., Girshick, R., Farhadi, A., 2016. You Only Look Once: Unified, Real-Time Object Detection. arXiv preprint arXiv:1506.02640.
- [38] Redmon, J., Farhadi, A., 2016. YOLO9000: Better, Faster, Stronger. arXiv preprint arXiv:1612.08242.
- [39] Redmon, J., Farhadi, A., 2018. YOLOv3: An Incremental Improvement. arXiv preprint arXiv:1804.02767.
- [40] ROBOTIS, 2017. Roboplus 1.0. [Online; accessed 27-February-2018].  
URL [http://en.robotis.com/model/board.php?bo\\_table=print\\_en&wr\\_id=48](http://en.robotis.com/model/board.php?bo_table=print_en&wr_id=48)
- [41] ROBOTIS e-Manual, 2016. AX-18F / AX-18A. Version 1.29.00, [Online; accessed 27-February-2018].  
URL [http://support.robotis.com/en/product/actuator/dynamixel/ax\\_series/ax-18f.htm](http://support.robotis.com/en/product/actuator/dynamixel/ax_series/ax-18f.htm)
- [42] ROBOTIS e-Manual, 2016. USB2Dynamixel. Version 1.29.00, [Online; accessed 26-February-2018].  
URL [http://support.robotis.com/en/product/auxdevice/interface/usb2dyl\\_manual.htm](http://support.robotis.com/en/product/auxdevice/interface/usb2dyl_manual.htm)
- [43] ROBOTIS e-Manual, 2017. MX-106T / MX-106R. Version 1.31.30, [Online; accessed 27-February-2018].  
URL [http://support.robotis.com/en/product/actuator/dynamixel/mx\\_series/mx-106.htm](http://support.robotis.com/en/product/actuator/dynamixel/mx_series/mx-106.htm)
- [44] ROBOTIS e-Manual, 2017. MX-28T / MX-28R / MX-28AT / MX-28AR. Version 1.31.30, [Online; accessed 27-February-2018].  
URL [http://support.robotis.com/en/product/actuator/dynamixel/mx\\_series/mx-28at\\_ar.htm](http://support.robotis.com/en/product/actuator/dynamixel/mx_series/mx-28at_ar.htm)

- 
- [45] ROBOTIS e-Manual, 2017. MX-64T / MX-64R / MX-64AT / MX-64AR. Version 1.31.30, [Online; accessed 27-February-2018].  
URL [http://support.robotis.com/en/product/actuator/dynamixel/mx\\_series/mx-64at\\_ar.htm](http://support.robotis.com/en/product/actuator/dynamixel/mx_series/mx-64at_ar.htm)
- [46] Spong, M. W., Hutchinson, S., Vidyasagar, M., 2006. Robot Modeling and Control. Wiley.
- [47] The University of Edinburgh, 2012-2016. Open On-line Course - AI Planning. [Online; accessed 11-June-2018].  
URL <http://www.aiai.ed.ac.uk/project/plan/ooc/>
- [48] Wibler, B. M., 1972. A Shakey Primer. Standford Research Institute.
- [49] Wolfe, J. M., Kluender, K. R., Levi, D. M., Bartoshuk, L. M., Hertz, R. S., Klatzky, R., Lederman, S. J., Merfeld, D. M., 2015. Sensation & Perception. Sinauer Associates.
- [50] Zeiler, M. D., Fergus, R., 2014. Visualizing and Understanding Convolutional Networks. ECCV.
- [51] Zeiler, M. D., Fergus, R., 2015. Striving for Simplicity: The All Convolutional Net. Accepted as a workshop contribution at ICLR 2015.