JACOBS
UNIVERSITY

School of Engineering and Science
Electrical Engineering and Computer Science

Bachelor's Thesis

A Cheap Chess Robot : Planning and Perception

Billy Okal
Oliver Dunkley

June 2, 2011

Supervisor: Prof. Dr. Andreas Nüchter

Abstract

Development of robots capable of playing chess is historically known to be a complex task and has been investigated for many years. Significant improvements have been made especially after the first computer program beat world human champion in the game over a decade ago. A number of robots of varying types have been developed in a bid to solve the problem, and in most cases additional constraints have been imposed on the game to allow the robots play the game. These modifications fundamentally alter the 'taste' of the game are are therefore not desired. In this thesis, we extend on some of the methods developed so far and relax on some of the said constraints. We run experiments on a low cost 5 DOF robot arm aided by a simple webcam using modular and flexible software that we build, and evaluate and compare the results with past findings in order to draw conclusions about applicability of certain fundamental algorithms in planning and perception in robot chess scenario and possibly develop new methodologies for solving the problem. Our results indicate the some of the said constraints can indeed be relaxed without sacrificing the game.

Table 1: Symbols and Abbreviations

DOF	Degrees of freedom of a mechanical system
IK	Inverse Kinematics
FK	Forward Kinematics
ROS	Robot Operating System
$\mathbb{N}$	Set of natural numbers
$\mathbb{R}$	Set of real numbers
$\mathcal{C}$ or *C-space*	Configuration space of a robot
$\mathcal{C}_{free}$	Valid Configuration space
${}^{b}_{a}T$	Transform of b with respect to frame a
${}^{b}_{a}P$	Translation of b with respect to frame a
${}^{b}_{a}R$	Rotation of b with respect to frame a
I_x	Identity Matrix of size x
BPS	Bits Per Second
EEF	End Effector
RRT	Rapidly-Exploring Random Trees
BiRRT	Bi-directional Rapidly-Exploring Random Trees
τ	Rapidly-Exploring Random tree
q	node, generally representing a joint configuration
SAC	Smart Arm Controller Node
ORN	The OpenRave Node
MCN	The Manipulation Core Node

Contents

1 Introduction **1**
1.1 Problem Definition 1
1.1.1 Goal 1
1.1.2 Assumptions and Restrictions 2

2 State of the Art **5**
2.1 Chess Playing Robots 5
2.1.1 Planning 6
2.1.2 Perception 6
2.2 Motivation 7

3 Methodology and Implementation **9**
3.1 Introduction 9
3.2 Core Control Unit 10
3.2.1 State Machine 10
3.2.2 Chess Engine 12
3.3 Perception 14
3.3.1 Chessboard Detection 14
3.3.2 Projection of the View 17
3.3.3 Detecting Chessboard Changes 18
3.3.4 Detecting Chess Moves 20
3.4 Manipulation 21
3.4.1 Frames and coordinate systems 22
3.4.2 Transforms 23
3.4.3 Important Functions 25
3.4.4 OpenRave 27
3.4.5 FK and IK 28
3.4.6 Planning and Trajectories 31
3.4.7 The Environment 39
3.4.8 The Manipulator 42
3.4.9 SAC - Smart Arm Controller Node 46

3.4.10 ORN - The OpenRave Node 47
3.4.11 MCN - The Manipulation Core Node 63

4 Experiments and Results **71**
4.1 Perception 71
4.1.1 Chessboard Detection 71
4.1.2 Projecting the View 71
4.1.3 Move Detection 71
4.1.4 Discussions 72
4.2 Manipulation 73
4.2.1 Experiments and Conclusions 73

5 Conclusion and Future Work **77**
5.1 Conclusion 77
5.2 Future Work 77

Chapter 1

Introduction

This research outlines the implementation of a low-cost chess playing robot, built entirely from cheap hobby-grade components. These components consist of a 5 DOF horizontally mounted manipulator (Section 3.4.8), a mount (Section 3.4.7), a webcam and standard chess components (Section 3.4.7). The focus of the investigation thus involves implementing a flexible and robust manipulator using the said components that is intelligent enough to understand when and which move a human player has made, and to respond by moving the appropriate figures on the chessboard. In this thesis, chess engines/algorithms are only discussed to the extent of how to incorporate them.

The interest for chess playing robots was sparked as early as 1770 when Wolfgang von Kempelen unveiled what appeared to be an automatic chess player, that would astonish and fascinate people all over the world, even after its fake nature was announced in the 1820s [17]. In our modern era, many chess playing robots exist. However, the majority of these either rely on modified chess boards and/or pieces to aid the robot sensors, or use fixed vision sensors with a fixed board or use industrial-grade manipulators. These restrictions simplify the problem at a cost of flexibility, generality and reproducibility. We propose an implementation that solely uses a single webcam as a sensor, a cheap robotic arm as an actuator and standard chess components. Furthermore, this implementation is modular and generic, allowing for easy interchanging of every major component: the visual processing, the camera, the chess engine, the manipulator and the motion planner.

1.1 Problem Definition

1.1.1 Goal

The goal is to implement a chess playing robot that humans can intuitively play against, without using any specialized chess equipment or manipulators. In order to achieve this, the robot implementation must satisfy each of the following sub-goals:

- Be able to move a chess figure from one location to another with an accuracy of $\pm 12\ mm$
- Recognize which move the opponent made, allowing it to track the state of the game
- Determine the 2d pose of board within an accuracy of $\pm 5\ mm$ and $\pm\ 4°$
- Determine the 2d location of each piece $\pm$ 5mm
- Completing a full move within 1 minute
- Use a webcam as the only sensor (excluding servo feedback)
- Total cost under EUR 1000 as of December 2010
- Integrate the sensor and actuator into a control loop that respects the rules of chess

Additionally, the following is also desirable:

- Visualization of the manipulator, chess components and environment
- Simulation of the manipulator and chess components
- Modularity - be able to swap out different visual processing algorithms, chess engines and motion planners with ease

1.1.2 Assumptions and Restrictions

To facilitate simplification of the problem without conflicting with goal, the following assumptions and restrictions apply:

- The human player shall make legal moves only
- The human player shall not interfere with the robot moves or the chess components unless it is his/her/its turn
- The board must be well (but can be arbitrarily) lit
- The board size is known
- The board can be moved, but only within the arms reachability
- Neighboring chess cells and each chess team have clear brightness differences
- Each figure's height is known
- The camera can be moved, but must have a clear and complete view of the board

- The figures are large enough that the manipulator can grab them, but small enough to not clutter the board
- The board is large enough to contain all the pieces with sufficient gripping space between them
- The figures are assumed to be cylindrical in shape

Chapter 2

State of the Art

Development of robots that are able to play a game of chess has drawn a lot of attention since the Deep Blue computer beat Gary Kasparov the then world chess champion [11], a feat considered a major breakthrough in artificial intelligence. Nevertheless, work on building such robots had been going on for years mostly geared towards solving specific parts of the problem. Such efforts had even been explored in classroom projects such as Groen's lab course in sensor integration [3] in which one such a robot was developed, albeit with many limitations. In effect, many approaches have been developed to tackle the problem, most of which focus on specific tasks such as vision and planning.

2.1 Chess Playing Robots

Most of the robots developed so far use simple arms with mostly between four and six DOF and the arm is usually mounted on a fixed base. Most of these systems have cameras mounted directly above the chessboard. A notable exception is the REEM-A humanoid robot developed by PAL Robotics which has 30 DOF [16]. The REEM-A uses stereo cameras and other sensors for acquiring environment information and expensive arms for manipulation. The MarineBlue [20] robot developed in 2003 is one of the few cheap and simplistic chess playing robots built so far. The MarineBlue was made of separate segments driven by servos, and aided by a high-quality camera fixed directly above the chessboard. Recently developed robots include Gambit [18] having a moderately costly 6 DOF arm and uses two cameras to interact with environment to execute table top manipulation. The Gambit system also uses modified chess pieces and chessboard to aid the vision system. More recently, the Association for Advancement of Artificial Intelligence (AAAI) and the International Conference on Robotics and Automation (ICRA) organized a small scale manipulation challenge with playing chess as the competition task [1]. Four robots participated and all involved expensive arms with exception of University of Alabama's entry and Chiara robot [7]. All together some of the distinctive properties of these robots are summarized in Table 2.1.

Table 2.1: State of the Art Chess Playing Robots

Robot	**Year**	**DOF**	**Sensors(Chess)**	**Remarks**
Gambit	2010	6	2 Cameras	Costs more than US$ 3000
REEM-A	2005	30	StereoCam	Humanoid, additional sensors
MarineBlue	2003	5	Sony DFW-VL500	costs US$ 550, modified chessboard
Chiara	2010	6	Logitech Webcam	Powered by 24 Dynamixel servos
Golem Chesster	2010	7	Unknown	Expensive Schunk arms

2.1.1 Planning

For most of the robots above, there no mention of extensive use of established planning algorithms or derivations thereof in the planning process, with exceptions for the expensive robots intended for general manipulation tasks like the REEM-A and Golem Chesster robots. Chess specific robots such as MarineBlue involve only few DOF, and most links can be easily related using simple equations when solving for IK. It is therefore easy to find analytical solutions when solving IK for such a robot. Nevertheless, solving for IK solutions is computationally intensive because of the huge number of possible solutions which have to be generated. The use of established motion planning algorithms [8] or derivatives gives a leeway for avoiding this computational cost by adapting the planners to take advantage of the workspace and respectively the *C-space*. This could also allow integration of multiple such algorithms, optimization of generated trajectories to make them smoother of generate new ones using techniques such as CHOMP [13] and even use of motion primitives [9] in the process.

2.1.2 Perception

The perception system used by most of the robots mentioned above largely involve a camera permanently mounted directly above the chessboard [3, 20]. Such a configuration greatly simplifies the vision problem in terms of avoiding occlusions among the pieces and reducing the perspective distortion to a minimum. Most of these systems also do not incorporate object recognition components, mostly because chess moves can still be determined based on prior information about the chessboard configuration. This also takes advantage of the full observability of the environment from a vision perspective. Some of these vision systems also restrict the lighting conditions of the environment in order to reduce complications in image processing. Furthermore, most systems use modified chessboards and well as carefully selected and/or designed chess pieces by choosing bright colors as witnessed in [1]. Allowing a non-stationary camera enables development of a

system that can easily be adopted to humanoid robots and mobile robots in general. This is also closer to emulating how human vision perceives the chessboard during the game. The requirement that a camera be placed directly above the chessboard can also be relaxed because there already exist efficient ways to correct the perspective distortion using homography/perspective projection and fast camera calibration. However, occlusions still need to be avoided.

2.2 Motivation

The motivation for this research is generally three-fold. The first goal is to improve upon the vision systems that have been developed for the robots mentioned above and develop a more robust and flexible system. Such a system shall use only one simple webcam which can be moved around during the course of the game as long as the camera still has a full view of the chessboard. We also intend to employ a standard chessboard piece set without modifications. We shall also investigate the effect of having an object recognition module in the vision system.

The second goal is to improve upon the planning systems used previously by integrating multiple motion planners and evaluating the outcome based on standard planner metrics such as the time it takes and the number of collision checks. We shall focus on the algorithmic approach to planning in order to build a system that is scalable and robust. We shall also make comparisons to come up with authoritative statements about which types of planners are suitable for the chess planning problem.

The last goal is to integrate the developments in the vision and planning systems into a modeling and control module which then monitors the robot during the game. We aim to make such a system robust, scalable and generic so that it can be easily ported to other robots with minimal changes. We shall employ modular software systems, Robot Operating System (ROS) [12] for the implementation of this system.

Chapter 3

Methodology and Implementation

3.1 Introduction

Overview In this chapter, we will discuss the implementation of the chess playing robot. The implementation is divided into three modules, "Core Control", "Perception" and "Manipulation" which are illustrated in Figure 3.1.

Core Control ties together the manipulation and perception pipelines and is responsible for the overall execution of playing chess. It requests which moves the user has made from the perception pipelines and responds by requesting the manipulation pipeline to execute a move. For a detailed description, see Section 3.2.

The perception pipeline is responsible for dealing with the task of observation. It detects when a user has moved and which move the user played, as well as the absolute location of the chess pieces. See Section 3.3 for a detailed description.

The manipulation pipeline is in charge of the task of executing chess moves. It safely moves chess pieces from starting positions to goal positions while avoiding collisions with other chess pieces, the environment and itself. To do this, it generates plans in a simulation which are then executed on the real hardware. For a detailed description, see Section 3.4.

ROS ROS is an open-source, meta-operating system. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. For an overview of ROS, check the the wiki `http://www.ros.org`

To facilitate modularity, the implementation described in this chapter uses the ROS framework. The following three sections will describe the modules listed above specifically. Each module consists of a set of independent processes (called ROS nodes) that communicate via ROS messages and services with one another to collaboratively play chess against a human. For each module, an interface was defined allowing for inde-

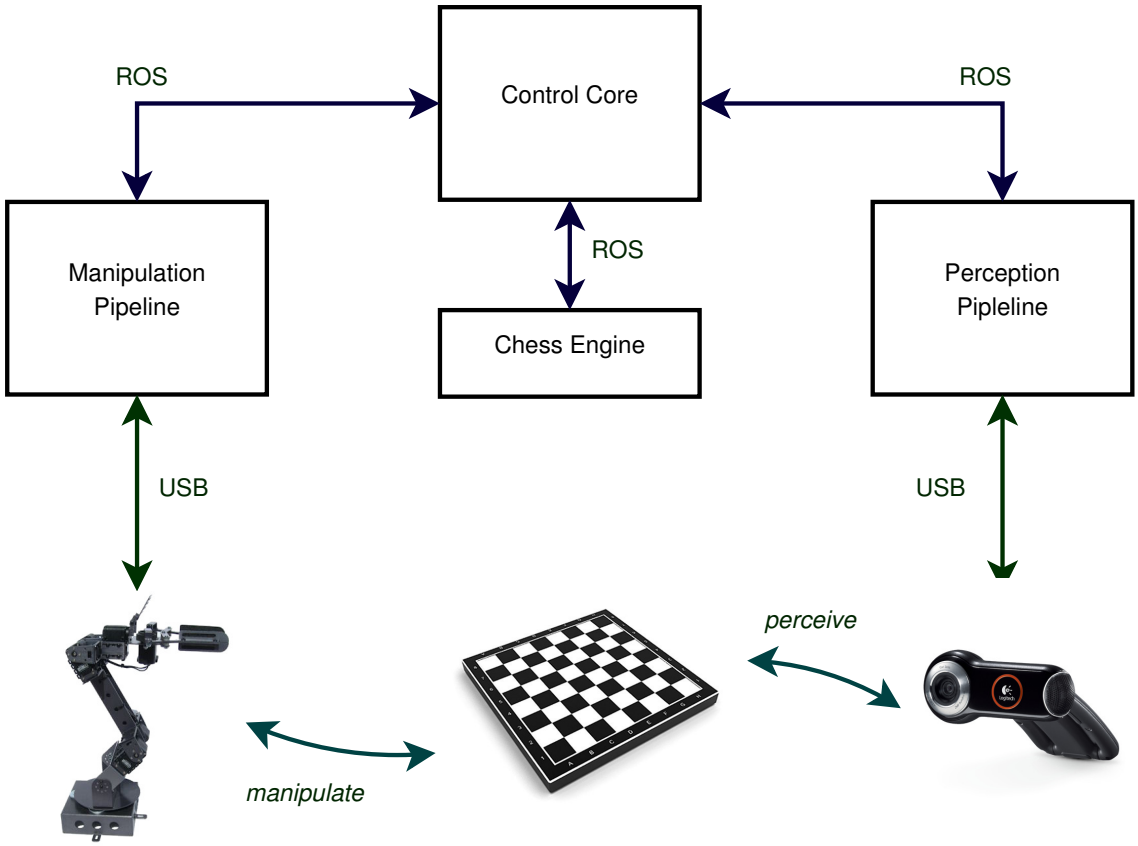


Figure 3.1: Implementation Components

pendent and modular development. Everything is run on an Ubuntu 10.10 Core2Duo machine.

3.2 Core Control Unit

Overview The Core Control Node (CCN) is solely responsible for organizing the control flow of the chess playing robot. It comprises of two nodes, the State Machine node and the Chess Engine node. They are both written in C++ and communicate via ROS. Below one can see a graphical overview in Figure 3.2.

3.2.1 State Machine

The state machine interfaces directly with the perception and manipulation pipeline, as well as as communicating with the chess engine. Below in Figure 3.3 one can see a

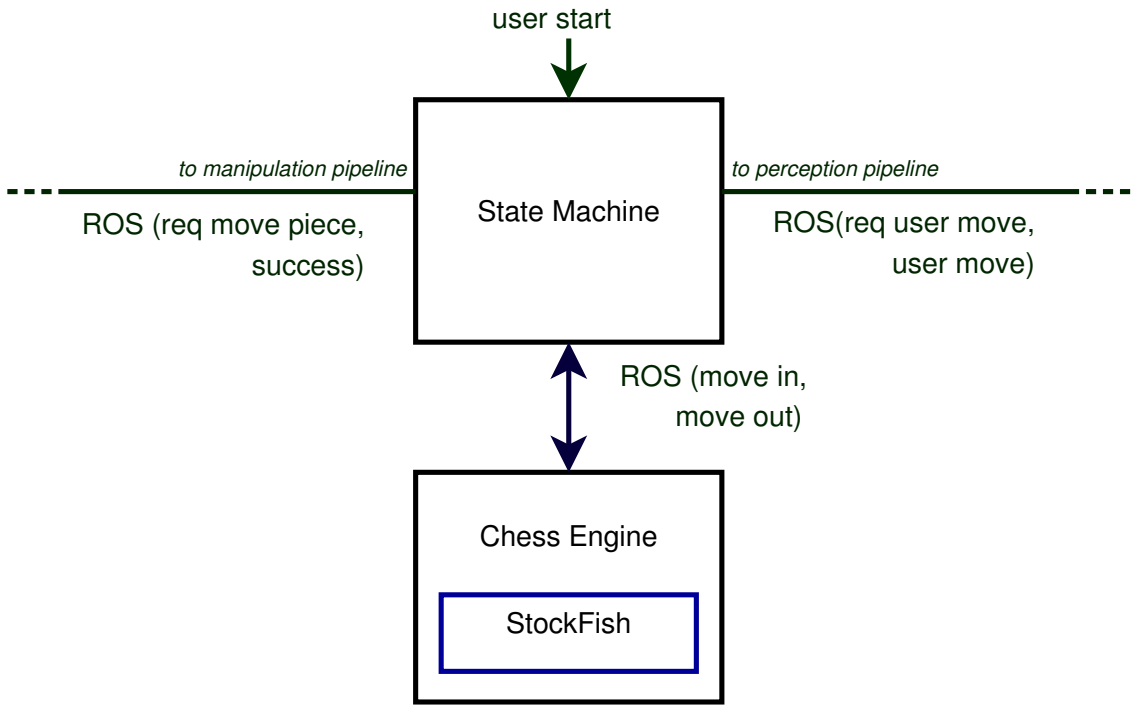


Figure 3.2: Core Control Overview

representation of its control flow.

The State Machine in Figure 3.3 clearly defines the start and end of a full game. Who starts is predetermined by the user. By default, the user is expected to start. After every move, the game state is evaluated for a potential draw or win scenario. When not in the start and end states, the Control Core is either waiting for a user move to be detected via the perception pipeline, or in the process of deciding which move to make with the chess engine, or executing its own move via the manipulation pipeline.

To illustrate the interactions between the components and the flow of execution, consider the following example: The robot has just finished executing a move, it is now the users turn. The Core Control Unit requests the users move from the perception pipeline. The Perception pipeline registers a user move A2 to A4, validates it and returns the move to the Core Control Unit, thereby simultaneously signaling it is now the robots turn, and the move the user performed. Using this information, the Control Core Unit queries the chess engine node for a response move. The engine responds with move B7 to A4, which also signals that the game is not in an end position (checkmate, etc). The Core Control Unit then requests the manipulation pipeline to execute the given move, by constructing the sequence of sub-moves required: as a capture is taking place, the figure at A4 must be removed before the figure at B7 can advance. Thanks to the vision pipeline, the exact

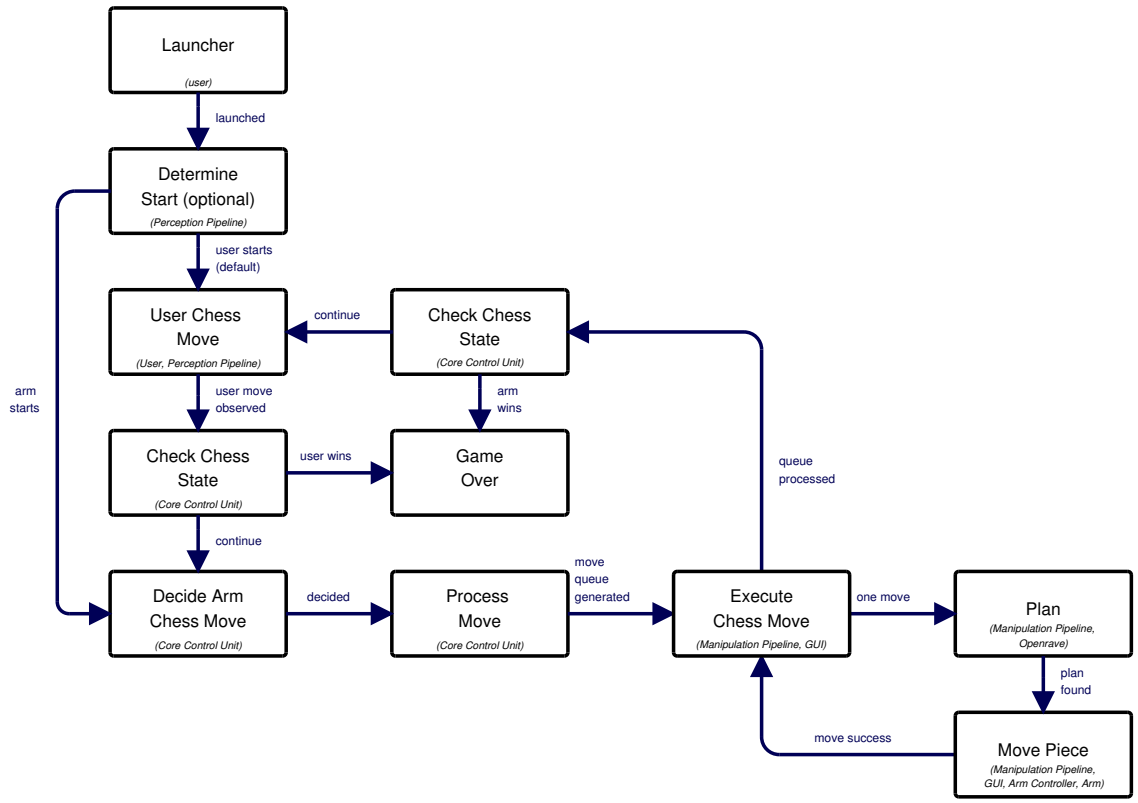


Figure 3.3: State Machine of Core Control Unit

coordinates of the figure to be moved and the goal coordinates in the world frame are known. The manipulation pipeline then takes the required steps to perform the first sub-move. Once this move is successfully executed, the second sub-move is executed, at which point the manipulation pipelines responds to the Control Core Unit that it has succeeded. The Control Core Unit then requests the users move from the perception pipeline and the cycle continues.

3.2.2 Chess Engine

Stockfish The chess engine node has two responsibilities:

1. Determining how to respond to a user move while obeying chess rules and hopefully, playing an effective game.

2. Signaling the Core Control Unit if the game has ended, so it can shut down the pipelines.

Stockfish was opted as the engine of choice, as it is an open source chess engine, using the UCI protocol and has existing binaries for Ubuntu 32/64 bit. With 3200+ ELO, it is considered very strong. Various computer chess rankings rate it as second or third behind the top gratis program Houdini and the commercial program Rybka[1].

In short, the node is a QT GUI exposed to the ROS network, hosting a chess engine it maintains on a separate process. It communicates using the UCI (Universal Chess Interface)[2] by setting up `stdin`/`stdout` pipes and reading/writing to these. The node keeps all parameters it requires to run the chess engine internally, and only exposes the minimal requirements to the ROS network (new game, user move, resulting move, game end), as shown in the `rxgraph` plot Figure 3.4 below.

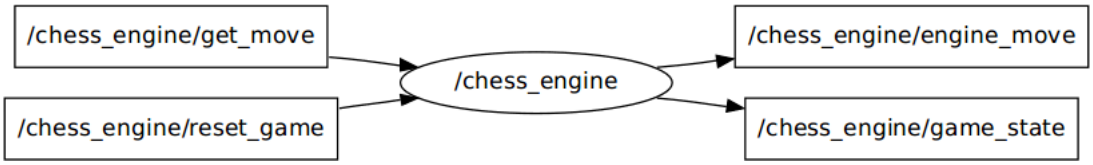


Figure 3.4: Chess Engine Node Ros Exposure

Using the GUI, one can manually write to `stdin` and read from `stdout` of the chess engine process. This supports debugging of the chess engine node. One can also force the node to emit specific response moves, which facilitates testing nodes dependent on the chess engine move decisions. A screen shot is shown in Figure 3.5.

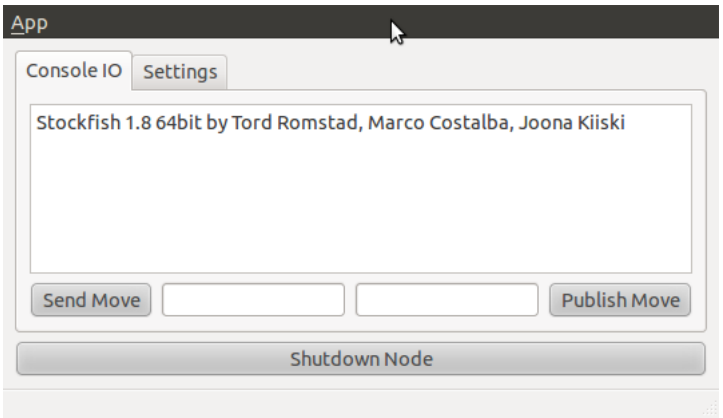


Figure 3.5: Chess Engine Node GUI

[1] `http://en.wikipedia.org/wiki/Stockfish_(chess)`
[2] `http://en.wikipedia.org/wiki/Universal_Chess_Interface`

3.3 Perception

The problem of robot chess perception can be divided into three main tasks which include: detection of the chessboard in a given image of the game scene; mapping the found chessboard view onto a 2D plane, and finally processing the image on the 2D plane to detect the changes on the chessboard and consequently the chess moves made. The complexity of this problem requires use of robust tools to be able to implement the given subtasks and also to integrate the overall solution efficiently. We therefore have four main modules for the overall task , three of which solve the subtasks and one module integrates the solutions. For this reason, we have used a number of state of the art software tools during the course of the investigation. These include; OpenCV (Open Source Computer Vision Library) [5], used for implementing the individual subtasks and ROS for integrating the solutions of the subtasks into a larger system solution. We mention the specific aspects of these tools used when describing the methodology with respect to the main subtasks in Section 3.3.1, Section 3.3.2, Section 3.3.3 and Section 3.3.4.

3.3.1 Chessboard Detection

Throughout the game, we 'observe' the scene using a simple webcam shown in Figure 3.6. At any given point that we want to analyze a user move, we grab a snapshot of the game scene at an arbitrary perspective. We then search for the chessboard in the snapshot image using the techniques summarized in 1 and compute the coordinates of the four corners of the chessboard detected in the scene if we detect one.

Figure 3.6: USB Webcam

Algorithm 1: Detecting a populated chessboard in a scene

Data: Raw Scene Image (img)
Result: Coordinates of the 4 corners of the chessboard
begin
  $img\_edges \longleftarrow EdgeDetect(img)$;
  $all\_lines \longleftarrow LineDetect(img\_edges)$;
  $MergeCloseLines(all\_lines)$;
  $bounding\_lines \longleftarrow FindBoundingLines(all\_lines)$;
  $corners \longleftarrow ComputerIntersections(bounding\_lines)$;
  return *corners*;
end

We first pre-process the raw image by converting it to grayscale in preparation for edge detection. The edge detection can be performed using any of the well established algorithms such as Canny [19], Sobel [19], Laplace [19] edge detectors among others. In our case, we make use of the Canny Edge Detector and its implementation in OpenCV `cv::canny`. The Canny edge detector operates on five simple steps as outlined below;

First, reduce noise by convolving the input image with a Gaussian filter. The larger the width of the filter, the lower the noise sensitivity of the detector. An example of such a filter is given below;

$$\frac{1}{115}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} \qquad \sigma = 1.4$$

Secondly, using a Sobel operator to perform a 2D spatial gradient measurement of the image in order to edge strengths. Sobel operator uses different masks for the x, G_x and y, G_y directions. The masks are given below;

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

The edge directions (θ) can then be computed using Eq. 3.1

$$\theta = \arctan\left[\frac{G_y}{G_x}\right] \tag{3.1}$$

Once the edge directions are known, they can be related to traceable edges in the image by checking the neighbor pixels. The tracing is done by setting all pixels not considered to be edges to a zero value leaving only the edge pixels as thin lines in the resultant

image. Finally edge contours that break above and below the threshold are removed by hysteresis which uses two thresholds.

The line detection can be performed by algorithms mainly based on the Hough transform defined below.

Definition The Hough Transform for a function/image $A(x, y)$ is defined as

$$H(\theta, \rho) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} A(x, y)\delta(\rho - x\cos\theta - y\sin\theta)\, dx\, dx \tag{3.2}$$

$$\text{where } \delta = \begin{cases} +\infty & x=0 \\ 0 & x\neq 0 \end{cases} \text{ and } \int_{-\infty}^{\infty} \delta(x)dx = 1$$

The key notion of the Hough transform [14] is the representation of lines in what is called *parameter space* or *image domain*. A line $y = mx + c$ is represented as shown in Figure 3.7.

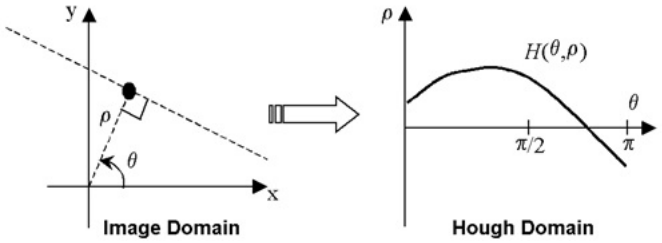


Figure 3.7: Image Domain and Hough Domain

This representation transforms each point in the original image A into a sinusoid given in Eq. 3.3 . The equation of the line can therefore be written as given in Eq. 3.4 . All points that lie on a line on the image will have their sinusoids cross a single point in the Hough domain. Since a back-projection operation is possible, we can put a threshold T on $H(\theta, \rho)$ so that the back-projection returns only lines with at least T points. This is typically done on binary images for which $H(\theta, \rho)$ gives a direct estimate of the number of points making a line.

$$\rho = x\cos\theta - y\sin\theta \tag{3.3}$$

$$y = \left\{ -\frac{\cos\theta}{\sin\theta} \right\} x + \left\{ \frac{\rho}{\sin\theta} \right\} \tag{3.4}$$

In our implementation we employ a well tested line detection method from OpenCV called `cv::HoughLines` which returns all the detected lines in an array. We then iterate over all the lines and merge those lines that are arbitrarily close to each other based on

distance between corresponding points on the lines. With the merged lines, we again iterate over all lines and search for extreme lines corresponding to bounding lines. This approach requires that the background on which the chessboard is placed be 'plain' so that the extreme lines correspond to the edges of the chessboard. This procedure identifies the four bounding lines and returns these in an array, from which we compute the intersections of these four lines with each other using standard linear algebra to determine the four corners of the chessboard.

3.3.2 Projection of the View

Using the coordinates of the four corners on the chessboard given by executing 1, we then project the chessboard on to a 2D plane by performing a perspective transformation. The transformation involves multiplication with a homography matrix. The homography matrix can be computed from a set of points by solving a system of linear equations. The problem can be stated as;

Given a 2D point $\vec{p} = (x, y)$ on the image plane, find a matrix H such that the point $\vec{p'} = (x', y')$ given by $\vec{p'} = H\vec{p}$ lies on a 2D plane P^2 of choice. Usually these points are then represented as 3D vectors using homogenous coordinates as shown in Eq. 3.5 and Eq. 3.6.

$$\vec{p} = (a, b, c), \quad s.t \quad x = \frac{a}{c},\ y = \frac{b}{c} \ \ and\ c \neq 0 \tag{3.5}$$

$$\vec{p'} = (a', b', c'), \quad s.t \quad x' = \frac{a'}{c'},\ y' = \frac{b'}{c'} \ \ and\ c' \neq 0 \tag{3.6}$$

Since the mapping from $P^2 \rightarrow P'^2$ can only be a projection if and only if it is an invertible mapping from P^2 to itself such that any set of points that line a line before the projection, still lie on a line after the projection [4]. H therefore must be a non-singular 3×3 matrix so that the 3D vectors can be related as;

$$\begin{bmatrix} a' \\ b' \\ c' \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} \tag{3.7}$$

This system can be solved using the DLT (Direct Linear Transform) algorithm [4]. For a given set of 2D point correspondences, we can expand Eq. 3.7 and normalize with respect to the homogenous components (c, c') to arrive at Eq. 3.8 and Eq. 3.9.

$$a'_i = \frac{h_1 a_i + h_2 b_i + h3}{h_7 a_i + h_8 b_i + h9} \tag{3.8}$$

$$b'_i = \frac{h_4 a_i + h_5 b_i + h6}{h_7 a_i + h_8 b_i + h9} \tag{3.9}$$

We can re-arrange Eq. 3.8 and Eq. 3.9 to arrive at two equations that are linear in the elements of H, meaning that every point correspondence yields two equations. To solve for the full matrix, we therefore need four point correspondences. We therefore get eight linearly independent equations given by Eq. 3.10.

$$\begin{bmatrix} a_1 & b_1 & 1 & 0 & 0 & 0 & -a_1a'_1 & -b_1a'_1 \\ a_2 & b_2 & 1 & 0 & 0 & 0 & -a_2a'_2 & -b_2a'_2 \\ a_3 & b_3 & 1 & 0 & 0 & 0 & -a_3a'_3 & -b_3a'_3 \\ a_4 & b_4 & 1 & 0 & 0 & 0 & -a_4a'_4 & -b_4a'_4 \\ 0 & 0 & 0 & a_1 & b_1 & 1 & -a_1b'_1 & -b_1b'_1 \\ 0 & 0 & 0 & a_2 & b_2 & 1 & -a_2b'_2 & -b_2b'_2 \\ 0 & 0 & 0 & a_3 & b_3 & 1 & -a_3b'_3 & -b_3b'_3 \\ 0 & 0 & 0 & a_4 & b_4 & 1 & -a_4b'_4 & -b_4b'_4 \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \end{bmatrix} = \begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \\ a'_4 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \end{bmatrix} \tag{3.10}$$

Which is then solved using standard numerical algorithms. In our implementation, we make use of certain handy functions in OpenCV namely, `cv::getPerspectiveTransform` which takes the four point correspondences and returns a 3×3 matrix. We then project the chessboard image on to a 2D plane by again using another handy OpenCV method called `cv::warpPerspective` which takes the original board image, the matrix and a size and returns a new image projected according to the matrix provided.

3.3.3 Detecting Chessboard Changes

In order to detect changes on the chessboard and consequently the chess moves made, we analyze successive chessboard images. We compare difference images and utilize information about the state of the game to determine changes on the chessboard. The analysis process is summarized into 2.

Starting with a difference image of successive scene observations, we remove noise by performing two morphological operations on the image in succession. These operations are defined as given.

Definition Let A and B be sets in Z^2. Further, let the sets have components $a = (a_1, a_2)$ for set A and $a = (b_1, b_2)$ for set B, then;

Translation of a set A by $z = (z_1, z_2)$ denoted as $(A)_z$ is defined as;

$$(A)_z := \{c | c = a + z, \quad for \quad a \in A\} \tag{3.11}$$

Reflection of a set A denoted by $\hat{A}$ is defined as;

$$\hat{A} := \{x | x = -a, \quad for \quad a \in A\} \tag{3.12}$$

Algorithm 2: Detect Changes on the Chessboard

```
Data: A Difference Image (img)
Result: A list of changes (change_string)
begin
    change_string ⟵ Null;
    thresh ⟵ some_value;
    erode(img);
    dilate(img);
    img_dt ⟵ distance_transform(img);
    large ⟵ find_maximum(img_dt);
    while large ⩾ thresh do
        locate_coordinates_of_blob_at_large;
        map_coordinates_to_board_dimensions;
        append_result_to_change_string;
        flood_fill(img, large);
        img_dt ⟵ distance_transform(img);
    return change_string;
end
```

Erosion of a set A by set B the mask, denoted by $A \ominus B$ is defined as;

$$A \ominus B := \{x \;\; s.t. \;\; (B)_x \subseteq A\}$$

Dilation of a set A by set B the mask, denoted by $A \oplus B$ is defined as;

$$A \oplus B := \{x | (B)_x \cap A \neq \emptyset\}$$

The implementation of the erosion and dilation is done using handy OpenCV methods, namely `cv::erode` and `cv::dilate` respectively.

The Distance Transform procedure labels each pixel in the image with the distance to the closest background pixel and can be computed using a number of established algorithms. Most of the algorithms differ mainly in the metrics used in the distance computation. For our purposes, we do not care which of these are used, as the simplest of these, the Euclidean Distance Transform works well for our needs. For the implementation, we employ a method in OpenCV called `cv::distanceTransform` which takes a binary image and returns another image with the distances. To find the pixel with the largest distance to the background pixels, we employ 3. The algorithm utilizes simple routines which locate pixels and retrieve their distances. In our implementation we represent a pixel by a `struct` hence making these routines simple one line statements of code.

Once the pixel is found, we infer the existence of a blob centered on the pixel whose size depends on the pixel's distance to background pixels. This information allows us

Algorithm 3: Find pixel with largest distance to background

```
Data: Labelled Image (img)
Result: Pixel
begin
    pixel ⟵ GetPixelAt(0, 0);
    for i ← 0 to GetRows(img) do
        for j ← 0 to GetColumns(img) do
            if GetDistance(GetPixelAt(i, j)) > GetDistance(pixel) then
                pixel ⟵ GetPixelAt(i, j)
    return pixel;
end
```

to associate the blob with a change on the chessboard. We map the pixels coordinates to 'chessboard dimensions (a – h, 1 – 8)' by first subtracting an offset from the pixel's coordinates, after which we divide the result by eight and these map directly to the chessboard squares. The exact offset is found by repeated tuning. After mapping the coordinates, we fill the blob with background color using OpenCV method `cv::floodFill` and repeat the procedure for all blobs of 'reasonable size'.

3.3.4 Detecting Chess Moves

In order to find out which chess pieces moved to which location, we analyze the change strings from Section 3.3.3 and compare against previous chessboard configuration to determine moves. We keep the chessboard configuration state in a 8×8 matrix of binary values $0 = empty, 1 = occupied$, and update this matrix at each step. Using this matrix we can infer whether a change say **a2b3** is a move from cell **a2** to cell **b3** or the reverse by comparing the current cell status with the previous status. We iterate over the change string, each time treating a pair of changes at a time.

With all the changes now converted into chess moves, we now generate poses for use in planning for manipulation of the chess pieces. For each move we generate a start pose and a goal pose in free space defined by a 3D vector $\vec{v} = (x, y, z)^T$ representing position and a quaternion $\vec{q} = (x, y, z, w)^T$ representing the orientation of the piece. For simplicity and because we do not have an object recognition system in our implementation, the quaternion component is always the default quaternion $\vec{q} = (0, 0, 0, 1)^T$ assuming that all pieces stand 'straight' on the board. Using the coordinates of the blob centers from Section 3.3.3, we compute the x and y components of the 3D position vector using information about the size of the chessboard squares (in our case $40mm \times 40mm$). The z component is again for simplicity assumed constant. Providing all these values albeit

using 'default ones' allows flexibility in the planner design as we then employ a generic pose and orientation representation scheme.

3.4 Manipulation

Overview The Manipulation Pipeline comprises of three ROS nodes and a manipulator. Figure 3.8 gives a brief overview.

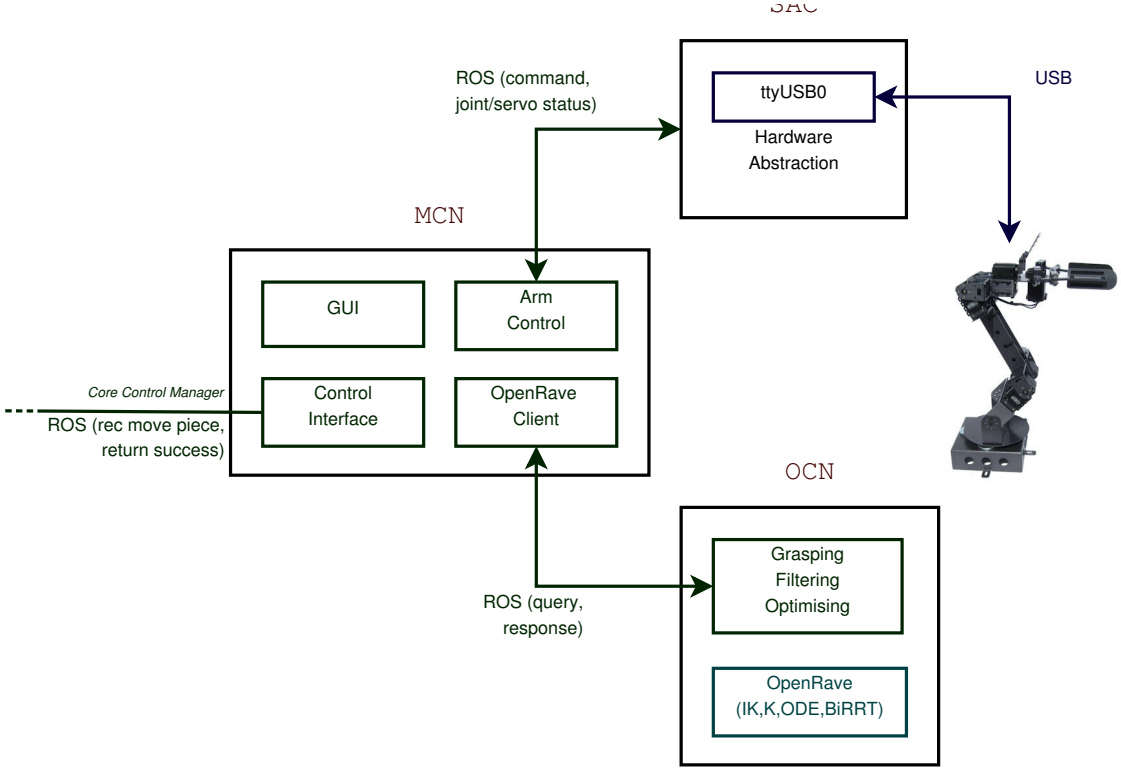


Figure 3.8: From top to bottom one can identify the SAC, MCN and ORN

In the center, one can see the main node which we will refer to as the "Manipulation Core Node" or "MCN" for short. In brief, it is responsible for organizing, visualizing and executing all manipulation related tasks. It boasts a heavy GUI where all parameters and configurations can be altered, (pre)viewed, saved and experimented with. It is the only node within the Manipulation Pipeline that provides services to the Core Control Node. For more details, see Section 3.4.11.

Above the MCN is the Smart Arm Controller (SAC) developed by the U.A. Robotics Research Group. Its purpose is to control the actual hardware and monitor the servo and joint states. More details in Section 3.4.9.

Under the MCN is the OpenRAVE Node (ORN). Using OpenRAVE (See ssec:or), the node is responsible for all manipulation calculations, algorithms, plan generation, IK solving, collision checking, etc. It also simulates the environment and is the heart of the Manipulation Pipeline. In this implementation, it is used as a server where the MCN can request plans or IK solutions from it. Described in detail in Section 3.4.10

The MCN relies on the SAC to control the hardware and on the ORN for more intelligent behavior. However, it depends on neither. A user can control and monitor the hardware without the ORN or can visualize the environment and generate plans without the SAC. This modular nature aids testing and development as the SAC and ORN are completely decoupled.

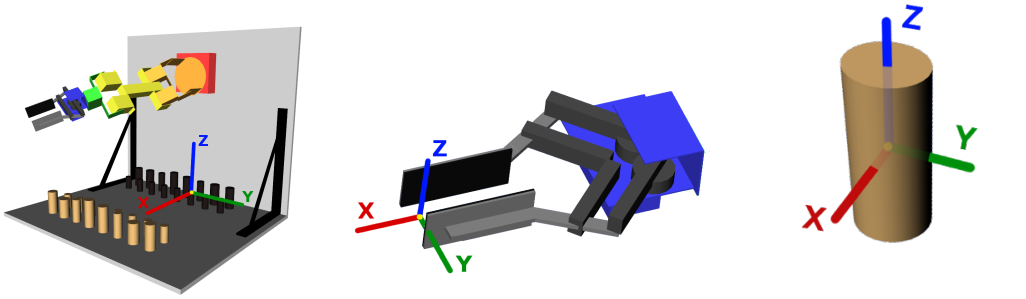
3.4.1 Frames and coordinate systems

Before we continue, we must define our coordinate systems. Three coordinate systems are of special interest that will be used to give a frame of reference to describe the transform of objects in them: the world frame ref, the end effector's frame eef and an object frame obj, which are all illustrated in 3.9 below. Generally, the X-axis will point forward, the Y-axis to the left and and the Z-axis upwards.

The world frame's origin lies at the intersection of the floor and wall panels, halfway along the width, on the ground and will be denoted as ref, for reference frame. See 3.9a.

The end effector frame's origin lies in line with the X axis that defines the wrist rotation, in front of the gripper such that the gripper closes around the EEF's origin's location. This is useful, as it allows us to grip objects by putting them in the origin of this frame. This frame will be denoted as eef. See 3.9b.

Each object has its own frame, where the origin lies in its center of the object. Note, that the only objects of relevance here are chess pieces, which are approximated as cylinders. This frame will be referenced as obj. See 3.9c



(a) World frame: ref **(b)** EEF frame: eef **(c)** Object frame: obj

Figure 3.9: Overview of frame coordinate systems and origins

3.4.2 Transforms

We describe the 3 DOF translational displacement (up/down, left/right, forward/backward) and 3 DOF rotational displacement (pitch, yaw, roll) of rigid bodies relative to the reference coordinate systems described above with the use of rotation matrices and translation vectors respectively. For a full 6 DOF description, we use the homogeneous transformation matrices.

Rotation A rotation matrix describes the relative orientation of an object to a reference frame. The columns of the following matrix consist of the unit vectors along the axes of the object, relative to the reference frame. We denote the relative orientation of an object frame *obj* with respect to a reference frame ref as:

$$ {}^{obj}_{ref}R = \begin{pmatrix} {}_{ref}x^{obj} & {}_{ref}y^{obj} & {}_{ref}z^{obj} \end{pmatrix} = \begin{pmatrix} x^{obj} \cdot x^{ref} & y^{obj} \cdot x^{ref} & z^{obj} \cdot x^{ref} \\ x^{obj} \cdot y^{ref} & y^{obj} \cdot y^{ref} & z^{obj} \cdot y^{ref} \\ x^{obj} \cdot z^{ref} & y^{obj} \cdot z^{ref} & z^{obj} \cdot z^{ref} \end{pmatrix} $$

Using this, we can derive the elementary rotations around the reference frame axes:

Roll, rotation around the X axis $R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix}$

Pitch, rotation around the Y axis: $R_y(\beta) = \begin{pmatrix} \cos(\beta) & 0 & -\sin(\beta) \\ 0 & 1 & 0 \\ \sin(\beta) & 0 & \cos(\beta) \end{pmatrix}$

Yaw, rotation around the Z axis: $R_z(\gamma) = \begin{pmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{pmatrix}$

The roll, pitch and yaw rotations can be used to place a 3D body in any orientation. A single rotation matrix can be formed by multiplying these rotation matrices to obtain:

$$ R(\gamma,\beta,\alpha) = R_z(\gamma)\, R_y(\beta)\, R_x(\alpha) = \begin{pmatrix} \cos\gamma\cos\beta & \cos\gamma\sin\beta\sin\alpha - \sin\gamma\cos\alpha & \cos\gamma\sin\beta\cos\alpha + \sin\gamma\sin\alpha \\ \sin\gamma\cos\beta & \sin\gamma\sin\beta\sin\alpha + \cos\gamma\cos\alpha & \sin\gamma\sin\beta\cos\alpha - \cos\gamma\sin\alpha \\ -\sin\beta & \cos\beta\sin\alpha & \cos\beta\cos\alpha \end{pmatrix}. \quad (3.13) $$

Note that since ${}^{b}_{a}R$ is orthonormal :

$$ {}^{a}_{b}R = {}^{b}_{a}R^{-1} = {}^{b}_{a}R^{T} \quad (3.14) $$

Translation Translation is where an object is displaced along its reference frame's axis without any rotation. The translation vector ${}^{obj}_{ref}\vec{P} = (P_x P_y P_z)^T$ displaces an object

obj along the x, y, z axis of reference frame ref. To obtain the translation of object obj that lies in the frame b, in the reference frame ref, one simply adds the translation of obj in its frame to the translation of frame b in the reference frame ref:

$$ {}^{obj}_{ref}P = {}^{b}_{ref}P + {}^{obj}_{b}P. \tag{3.15} $$

Homogeneous Transform The coordinates of an object obj, relative to a frame b, rotated and translated with respect to a reference frame ref, are given by:

$$ {}^{obj}_{ref}P = {}^{b}_{ref}R\,{}^{obj}_{b}P + {}^{b}_{ref}P. \tag{3.16} $$

This can be compacted into the form of a homogeneous transformation matrix that is defined as:

$$ {}^{b}_{a}T = \begin{pmatrix} {}^{b}_{a}R & {}^{b}_{a}P \\ 0_{1\times 3} & 1 \end{pmatrix} \tag{3.17} $$

.

This matrix represents the orientation and position of a frame b, whose orientation relative to the reference frame a is described by the rotation matrix ${}^{b}_{a}R$ and whose origin, relative to the same reference frame a, is described by ${}^{b}_{a}P$ is, thus, the full 6 DOF representation of a frame in three-dimensional space. If the coordinates of an object obj are known with respect to a frame b, then its coordinates, relative to reference frame ref are found by:

$$ \begin{pmatrix} {}^{obj}_{ref}P \\ 1 \end{pmatrix} = {}^{b}_{ref}T \begin{pmatrix} {}_{b}P^{obj} \\ 1 \end{pmatrix} \tag{3.18} $$

Note that this is the same as (3.16).

The following examples illustrate how to change frames:

If for example object frame obj, is known, relative to end effector frame eef, whose pose is known with respect to the world frame ref, we can obtain the object in the world frame transform ${}^{obj}_{ref}T$ as follows:

$$ {}^{obj}_{ref}T = {}^{eef}_{ref}T\,{}^{obj}_{eef}T \tag{3.19} $$

This will later prove useful, when converting transforms in the eef frame to the ref frame. If one would like to go the other way, i.e. one is given the end effector frame eef and an object frame obj, both relative to the world frame ref, and would like obtain the transform of the object frame obj in the end effector frame eef, i.e. ${}^{obj}_{eef}T$, one uses the inverse or the transpose since (3.14):

$$ {}^{obj}_{eef}T = {}^{eef}_{ref}T^{-1}\,{}^{obj}_{ref}T = {}^{eef}_{ref}T^{-T}\,{}^{obj}_{ref}T \tag{3.20} $$

This will be used when we plan with objects that are grasped, i.e. attached to the gripper.

3.4.3 Important Functions

The most important and frequently used functions are defined below.

Line-Plane intersection angle

It will be of essential importance to extract the angle between ${}^{obj}_{ref}axis_Y$, ${}^{obj}_{ref}axis_X$ and ${}^{obj}_{ref}axis_Z$ of a given transform ${}^{obj}_{ref}T$ and the plane spanned by ${}^{ref}_{ref}axis_X$ and ${}^{ref}_{ref}axis_Y$, in order to later define "uprightness" and "levelness". To do this, we first identify the normal vector to the plane and the directional component of ${}^{obj}_{ref}T$ that represents the axis we wish to intersect the plane with:

$${}^{obj}_{ref}axis_X = \begin{pmatrix} {}^{obj}_{ref}T_{0,0} \\ {}^{obj}_{ref}T_{1,0} \\ {}^{obj}_{ref}T_{2,0} \end{pmatrix} \quad {}^{obj}_{ref}axis_Y = \begin{pmatrix} {}^{obj}_{ref}T_{0,1} \\ {}^{obj}_{ref}T_{1,1} \\ {}^{obj}_{ref}T_{2,1} \end{pmatrix} \quad {}^{obj}_{ref}axis_Z = \begin{pmatrix} {}^{obj}_{ref}T_{0,2} \\ {}^{obj}_{ref}T_{1,2} \\ {}^{obj}_{ref}T_{2,2} \end{pmatrix}$$

The normal vector of the plane spanned by ${}^{ref}_{ref}axis_X$ and ${}^{ref}_{ref}axis_Y$ is $\vec{plane}_{XY} = (0,0,1)^T$. Note that the four vectors need not necessarily be normalised. Let $\vec{P}$ be the plane vector and $\vec{A}_a$ an axis vector, where a denotes the axis. Then

$$\angle\left(\vec{P}, \vec{A_{axis}}\right) = \arcsin\left(\frac{|P_x A_{ax} + P_y A_{ay} + P_z A_{az}|}{\sqrt{P_x^2 + P_y^2 + P_z^2}\sqrt{A_{a_x}^2 + A_{a_y}^2 + A_{a_z}^2}}\right) = \arcsin\left(\frac{|P \cdot A_a|}{||P||\,||A_a||}\right) \tag{3.21}$$

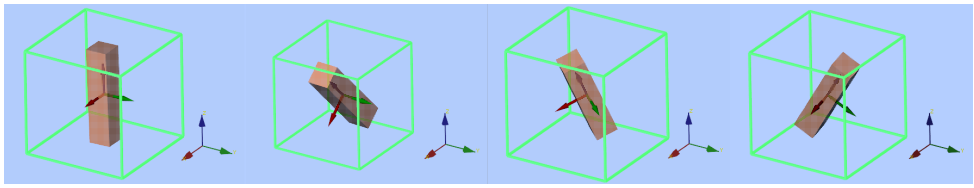
Levelness of a transform

Here we will define our notion of *levelness* and a function which evaluates how level a given transform is. This will be used in Section 3.4.10 to grasp an object with the EEF being as level as possible.

Intuitively, *levelness* describes how parallel to the floor an object is in the world frame, where parallelism is not affected by rotation around the objects local Y axis. This lets us describe parallelism in terms of the intersection angle between the objects local Y axis and the floor. For this, we make use of (3.21):

$$
\begin{aligned}
levelness({}^{obj}_{ref}T) &= \angle\left(\vec{P}, \vec{A}_y\right) \\
&= \angle\left(\begin{pmatrix}0\\0\\1\end{pmatrix}, \begin{pmatrix}{}^{obj}_{ref}T_{0,1}\\ {}^{obj}_{ref}T_{1,1}\\ {}^{obj}_{ref}T_{2,1}\end{pmatrix}\right) \\
&= \arcsin\left(\frac{|{}^{obj}_{ref}T_{2,1}|}{\sqrt{{}^{obj}_{ref}T_{0,1}{}^2 + {}^{obj}_{ref}T_{1,1}{}^2 + {}^{obj}_{ref}T_{2,1}{}^2}}\right)
\end{aligned} \tag{3.22}
$$

Given the transform of an object in the world frame ${}^{obj}_{ref}T$, function $levelness({}^{obj}_{ref}T)$ now returns the angle away from perfect levelness. Therefore, the lower the angle obtained, the more level the object is. Some examples of non-level and level objects are shown in 3.10.



(a) Level **(b)** Level **(c)** Non Level **(d)** Non Level

Figure 3.10: Examples of non level and level transforms of an object.

Uprightness of a transform

Here we define our notion of *uprightness*. Intuitively, uprightness is how little an object is "tipped over". For example, a glass of water sitting on a flat table is perfectly upright. To pour water out of the glass, one must tip it over somewhat (i.e. rotate it around its local Y and/or X axis), thereby reducing its uprightness. Maximising uprightness is synonymous with maximising the objects local Z axis and floor intersection angle, which is obviously bounded by 0° (lying on its side) and 90° (perfectly upright). Technically, this would mean a perfectly upside down glass would also be considered perfectly upright, as its Z axis intersects the floor at 90° too. Therefore, we will also introduce a function which evaluates whether the given transform of an object in the world frame is upside down or not. Using this, we can enlarge the interval from -90° (fully upside down) to 90° (fully upright). As we wish to use this as an error function, we will shift the interval such that 0° is perfectly upright and 180° is fully upside down.

First we must determine if an object transform is not upside down. Intuitively, we construct a transform that does nothing other than translate along its Z axis positively,

apply it to the transform of the object and then check whether the obtained transform lies over or under the object transform by comparing Z coordinates in the world frame:

$$isUpright({}^{obj}_{ref}T) = \begin{cases} 1 & \text{if } ({}^{obj}_{ref}T \cdot M)_{2,3} \geq {}^{obj}_{ref}T_{2,3} \\ 0 & \text{elsewise} \end{cases}$$

where M is the transform that goes one step in positive Z direction,

i.e. $M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$.

Now we can put it all together to define uprightness:

$$\begin{aligned} uprightness({}^{obj}_{ref}T) &= \left| 1.57^c - \angle\left(\vec{P}, \vec{A_z}\right) - \left(1.57^{cc} \cdot isUpright({}^{obj}_{ref}T)\right) \right| \\ &= \left| 1.57^c - \angle\left(\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} {}^{obj}_{ref}T_{0,2} \\ {}^{obj}_{ref}T_{1,2} \\ {}^{obj}_{ref}T_{2,2} \end{pmatrix} \right) - \left(1.57^c \cdot isUpright({}^{obj}_{ref}T)\right) \right| \\ &= \left| 1.57^c - \arcsin\left(\frac{|{}^{obj}_{ref}T_{2,2}|}{\sqrt{{}^{obj}_{ref}T_{0,2}{}^2 + {}^{obj}_{ref}T_{1,2}{}^2 + {}^{obj}_{ref}T_{2,2}{}^2}} \right) \right. \\ &\qquad \left. - \left(1.57^c \cdot isUpright({}^{obj}_{ref}T)\right) \right| \end{aligned} \tag{3.23}$$

This function now returns the angle that transforms are away from perfect uprightness, even if they are upside down. This will be used in Section 4.2.1 to place chess pieces as upright as possible.

3.4.4 OpenRave

Introduction OpenRAVE is the main manipulation framework we used in the manipulation pipeline. It stands for Open Robotics Automation Virtual Environment and is maintained by Rosen Diankov. It provides basic simulation and visualisation, transform querries of loaded kinematic bodies and kinematic chain link, IK solution generation, planning, collision check querries (ODE) and many other things. How it works and what exactly it provides is not scope of this thesis. For more information see [2]. For more information on how we used the viewer and simulation see Section 3.4.10.

IK solver generation OpenRAVE automatically generates a closed form analytic IK solver for a given kinematic structure. For the ones we used, please see Section 3.4.5.

Planning OpenRAVE has many inbuilt planners. We stick with the default one called BiRRT that is based on the RRT-Connect algorithm outlined in Section 3.4.6.

3.4.5 FK and IK

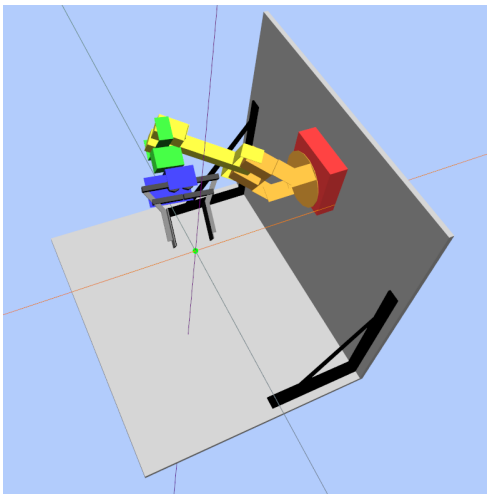
FK Forward kinematics is the computation of the transform of an element of a kinematic chain. In our case the element is usually the EEF (and occasionally the object it has grasped) and the kinematic chain are the links and joints that the arm is comprised of. In short, given the manipulators joint angles, forward kinematics calculates the EEF transform. Most of the time we rely on OpenRAVE (see Section 3.4.4) to do all FK calculations for us, except for when we manually add a grasped object to the kinematic chain of the manipulator. To do this, we use 3.20 to calculate ${}^{obj}_{eef}T$ from ${}^{obj}_{ref}T$ and ${}^{eef}_{ref}T$. This gives us the transform that takes us from the EEF to the grasped object. Then we can simply multiply ${}^{obj}_{eef}T$ with ${}^{eef}_{ref}T$ at any time to obtain ${}^{obj}_{ref}T$, dependent on the current joint angles, as ${}^{eef}_{ref}T$ varies with different joint angles. Of course, every time we release and grasp the object again, ${}^{obj}_{eef}T$ needs to be recalculated as a new grasp might result in a new relationship between the EEF and the attached object. Later, knowing ${}^{obj}_{ref}T$ in terms of joint configurations will allow us to use 3.23 on it to select the joint angles that puts the grasped object as upright as possible.

IK Inverse kinematics is a little more complicated, as many or no solutions may exist. Inverse kinematics is the computation that finds joint angles of a kinematic chain that brings the desired link (e.g. the EEF or attached object) to a desired transform. OpenRAVE (see Section 3.4.4) can generate many different types of IK solvers for a given kinematic chain. In our case, the manipulator has 4 DOF, which severely limits the type of IK solver we can use. We will only use two IK solvers:

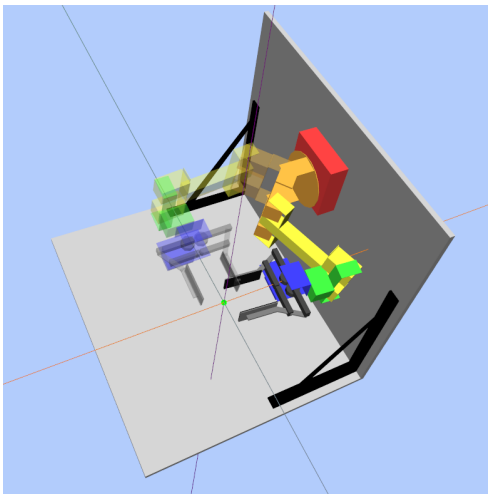
- Translation3D IK - given a goal translation ${}^{goal}_{ref}P$, calculate the required joint angles that places the desired link there. Useful for finding joint configurations that allow the manipulator to grasp an object at a specified position.
- TranslationLocalGlobal6D IK - given a goal translation ${}^{goal}_{ref}P$ and an offset translation ${}^{goal}_{link}P$, calculate the required joint angles that places the desired link (e.g. EEF) on the goal after the offset has been applied. This is useful to find joint configurations to release a grasped object at a specific goal. This IK solver did not exist prior to this project. I suggested it to the OpenRAVE maintainer Rosen Diankov who agreed it would be useful and implemented it very quickly.

Note that each of these IK solvers requires only 3 DOF. Therefore, the 4th DOF of the manipulator (i.e. the wrist rotation) is set as a free joint, meaning it is not used in the IK calculations. This intuitively makes sense as rotating the wrist does not change the

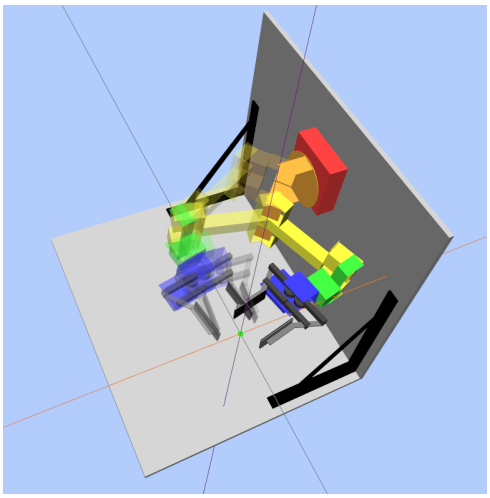
EEF translation. For every IK solution generated, infinite solutions exist as we can arbitrarily chose the 4th DOF value that is within the joint limits. To overcome this, we discretise this continuous space in 0.1^c intervals, limiting the number of solutions generated for each solution of the first 3 DOF to a maximum of $\lfloor \frac{Upperwristlimit - Lowerwristlimit}{0.1} \rfloor = \left\lfloor \frac{2.617-(-2.617)}{0.1} \right\rfloor = 52$ solutions. fig:ik shows Ik solutions for the given green point representing a translation. The first 4 are filtered to show one IK solution per IK solution of the first 3 DOF, while the last picture demonstrates all the IK solutions before discretisation.



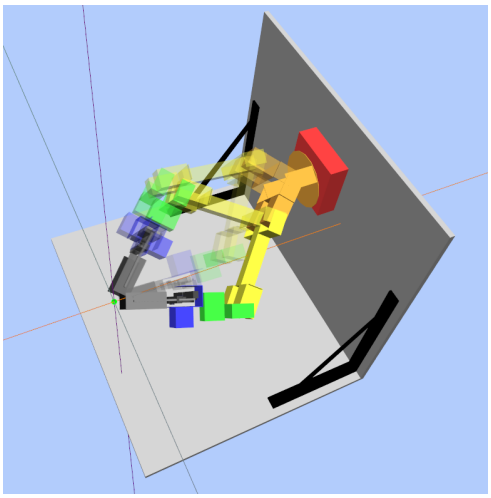
(a) One IK solution



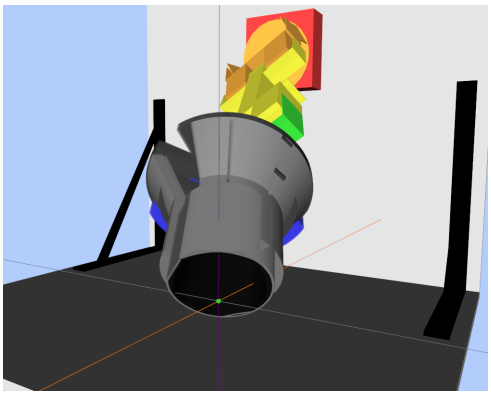
(b) Two IK solutions



(c) Three IK solutions



(d) Four IK solutions

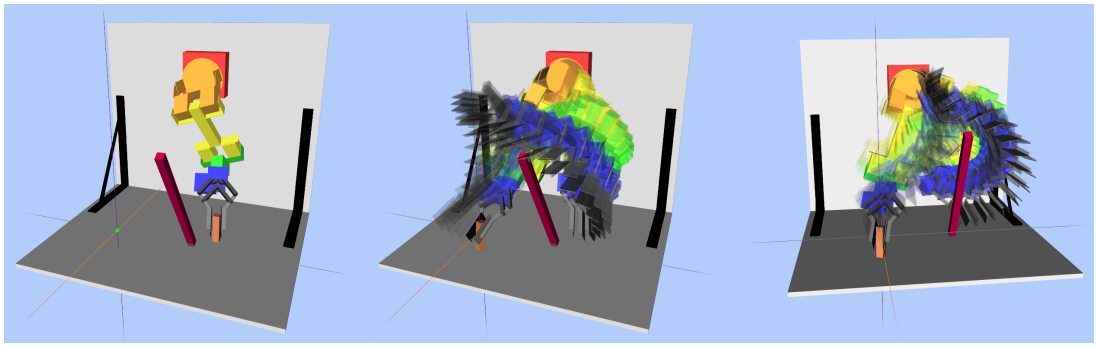


(e) All IK solutions

Figure 3.11: IK solutions for different translations. The first 4 only show IK where the wrist is fixed, while the last image demonstrates all IK solutions

3.4.6 Planning and Trajectories

Trajectory Trajectories are the results of planners. Valid trajectories consist of a list of valid manipulator joint configurations that incrementally vary in such a way that move the arm step by step from one joint configuration to another. In our case, this would be a list of vectors of length 5, one element for each DOF. If the incremental difference between each vectors respective elements are small enough, the arm will appear to move continuously. Intuitively, a trajectory is the "path" a manipulator takes to reach a goal configuration from an initial configuration. Usually, each incremental configuration also has an associated time stamp, thereby allowing the trajectory to accelerate and decelerate the arm along the course of execution. For simplicity, we omit this. Below in Figure 3.12 one can see every 10th joint configuration that brings the object to the specified translation without causing collisions or joint limit invalidation.



(a) Initial configuration **(b)** Trajectory generated **(c)** Different perspective

Figure 3.12: The resulting BiRRT plan: a trajectory to move the piece to the given goal translation. Every 10th joint configuration is plotted with decreasing transparency

Planning In our implementation planning can be seen as searching for valid trajectories that bring the manipulator from an initial configuration to a goal configuration while satisfying various navigation constraints, such as never violating the joint limits and never causing arm → arm (self) collisions or arm → environment collisions. Note that a grasped object is considered to be part of the arm.

There are many approaches to finding a solution in this complex search space, such as employing potential fields or sampling-based algorithms. OpenRAVE has various planners such as RRT, BiRRT, RA*, etc and we will use its default planner BiRRT, which grows two RRTs and attempts to connect them. RRTs and BiRRTs are both outlined in relevant context in Section 3.4.6 and Section 3.4.6 below. An example of the result of such a plan is shown in Figure 3.12. Before we can look at BiRRTs, we must first understand the underlying RRT algorithm and data structure.

RRT RRT stands for *Rapidly-Exploring Random Trees*, which is detailed in [10].

Briefly put, the RRT algorithm constructs a tree τ in $\mathcal{C}_{free}$ rooted at an initial configuration $q_{init} \in \mathcal{C}_{free}$ and incrementally grows itself until a continuous connection between q_{init} and a desired configuration $q_{goal} \in \mathcal{C}_{free}$ is found, at which point it returns the q_is$\in \mathcal{C}_{free}$ on the found path. To do this effectively, the construction of a RRT is biased towards unexplored portions of $\mathcal{C}$, meaning it will generally attempt to explore in the direction of the greatest unknown region at every iteration of construction. In our context, each node q_i of the tree τ represents a joint configuration and every edge a valid move from one configuration to another, meaning the path of q_is returned by the algorithm corresponds to a list of joint configurations which are successively near, i.e. a trajectory as described in Section 3.4.6.

[6] has shown that RRTs are probabilistically complete, meaning the probability that they will produce a solution increases with time spent, but they cannot determine if no solution exists. Therefore the runtime and/or maximum number of nodes is capped, and if the cap is reached the problem is deemed insolvable.

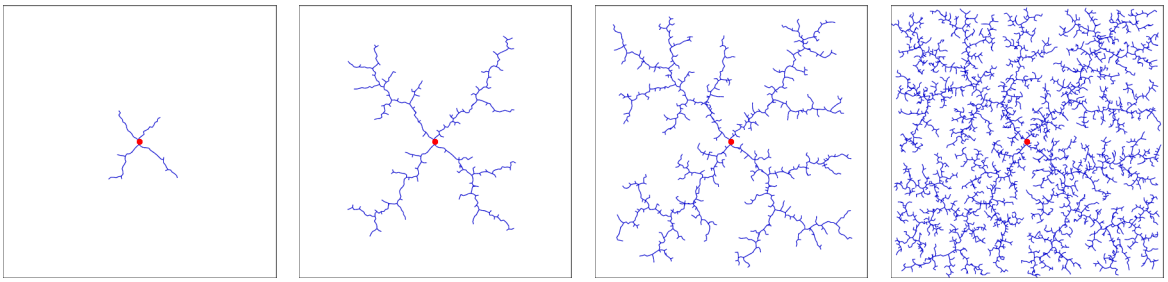
The basic RRT construction algorithm is given in algorithm 4.

Algorithm 4: BUILD(): Construct an RRT that eventually creates a path between q_{init} and q_{goal} if one exists

Input: Initial configuration q_{init}, goal configuration q_{goal}, number of nodes I_{max}, incremental distance ε
Output: Path $q_{init}, \ldots, q_{goal} \subseteq RRTtree\tau$
begin
  τ.INIT(q_{init}) ;
  for $i \leftarrow 1$ **to** I_{max} **do**
    $q_{rand} \leftarrow$ BIASED_RANDOM_CONFIG() ;
    if *EXTEND*(τ, q_{rand}, ε) = *Solved* **then**
      `// Return found solution`
      return τ.PATH(q_{init}, q_{goal});
  `// No solution found within` I_{max} `nodes`
  return No Solution;
end

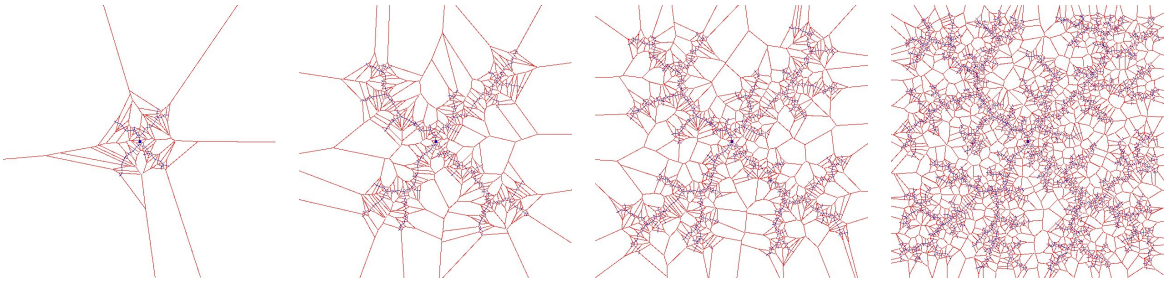
After the tree τ has been initialised with just its root q_{init}, nodes are iteratively added to τ in the following way: τ is extended towards a random but biasedly chosen node q_{rand} selected by `BIASED_RANDOM_CONFIG()`. This function selects a node to extend towards with probability proportional to the area of its Voronoi region (intuitively large Voronoi cells represent large unexplored areas), which causes the RRT to be biased towards rapid exploration before ultimately covering $\mathcal{C}_{free}$ uniformly. Illustration of the explorative behaviour is illustrated in Figure 3.13, where an RRT is applied to 2D square space and

is grown from the center. Figure 3.14 shows the corresponding Voronoi Cells.



(a) Initial Branching out **(b)** Rapidly exploring **(c)** Continued exploring **(d)** Uniformly coveraging the space

Figure 3.13: Four snapshots of an RRT expanding in 2D square space. Notice the exploratory behaviour and eventual uniform space convergence trend (from [6])



(a) Initial Branching out **(b)** Rapidly exploring **(c)** Continued exploring **(d)** Uniformly coveraging the space

Figure 3.14: Four snapshots of an RRT rapdily expanding biasedly towards large Voronoi regions which are indicated by red outlines (from [6])

Once q_{rand} is picked, the RRT τ then attempts to extend towards it using function `EXTEND()` outlined in algorithm 5. `EXTEND()` first determines q_{near}, the closest node to q_{rand} already in the RRT τ. The function `NEW_CONFIG()` steps in direction q_{rand} with a new node q_{new} with distance ε and returns q_{new} if $q_new \in \mathcal{C}_{free}$, i.e. is valid. One of four things can then happen:

Reached q_{rand} is directly added to RRT τ, as τ, already contains a node within ε of q_{rand}, i.e. $q_{rand} = q_{new}$

Advanced $q_{new} \neq q_{rand}$ indicating a successful but not final step towards q_{rand}. q_{new} then becomes a new node of RRT τ and an edge is added between q_{near} and q_{new}

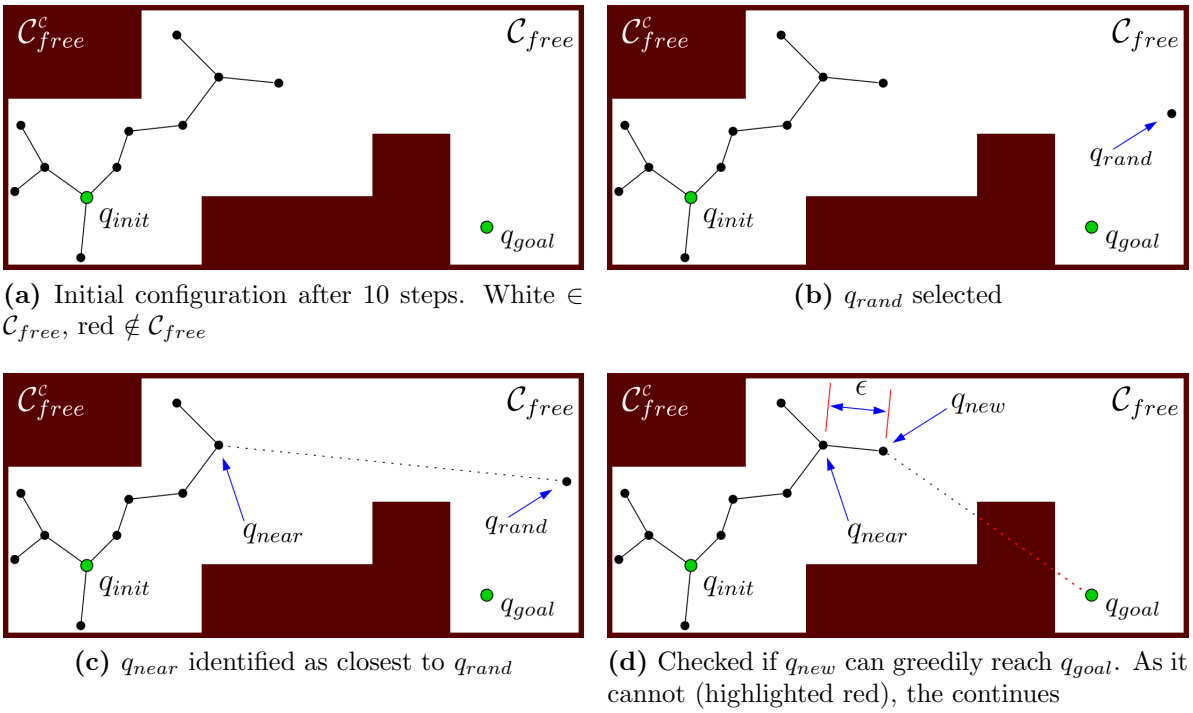
Trapped $q_{new} \notin \mathcal{C}_{free}$, i.e. the proposed new node is rejected due to invalidity

Solved After every successful `EXTEND()` operation, CHECK_FINISHED() checks in a greedy fashion whether one can directly reach q_{goal} from q_{new}. If this is the case, an edge from q_{new} to q_{goal} is added to RRT τ, which then successfully bridges q_{init} to q_{goal} resulting in a sequence of nodes that can be returned as the solution, and if not, the RRT continues to expand until a solution is found or the maximum number of nodes I_{max} is reached and the problem deemed insolvable.

The `EXTEND()` operation is illustrated in Figure 3.15. Note that this subsection on planning is to provide a general overview and idea of how RRTs work and can be applied to finding trajectories. Details, such as how to sample $\mathcal{C}$ and determine ε by using swept joint volumes to value more important joints higher are omitted. For exact details on how RRT is used and implemented see [2].

Algorithm 5: EXTEND(): Extend τ towards q_{rand} with distance ε and check if q_{goal} can be reached

Input: random node q_{rand}, goal node q_{goal}, an RRT tree τ, incremental distance ε
Output: Status
begin
  $q_{near} \leftarrow NEAREST\_NEIGBOUR(q_{rand},\tau)$;
  $q_{new} \leftarrow NEW\_CONFIG(q_{rand}, q_{near}, \varepsilon)$;
  if $q_{new} \neq NULL$ **then**
    τ.ADD_NODE(q_{new});
    τ.ADD_EDGE(q_{near}, q_{new});
    if *CHECK_FINISHED*(q_{new}, q_{goal}) **then**
      // q_{new} connects to q_{goal}, so we have a compelte path
      τ.ADD_EDGE(q_{new}, q_{goal});
      return *Solved*;
    else if $q_{new} = q_{rand}$ **then**
      // reached q_{rand}
      return *Reached*;
    else
      // extended towards q_{rand}
      return *Advanced*;
  else
    // $q_{new} \notin \mathcal{C}_{free}$
    return *Trapped*;
end



(a) Initial configuration after 10 steps. White $\in \mathcal{C}_{free}$, red $\notin \mathcal{C}_{free}$

(b) q_{rand} selected

(c) q_{near} identified as closest to q_{rand}

(d) Checked if q_{new} can greedily reach q_{goal}. As it cannot (highlighted red), the continues

Figure 3.15: The steps of the `EXTEND()` operation while building an RRT tree. Note that if the tree illustrated was one of the BiRRT trees, q_{goal} would be the closest node of the opposite tree.

BiRRT BiRRT stands for *Bi-directional Rapidly-Exploring Random Trees*, which uses the RRT-Connect algorithm detailed in [6] and was specifically implemeneted for manipulation planning in [2]. Intuitively, BiRRTs grow two RRTs, τ_I from initial node q_{init} and τ_G from goal node q_{goal} ,towards each other, utilising the rapidly exploring nature of RRTs to grow and a Connect Heuristic to join the trees, even over long distances. A connection between the two trees implies that a valid path has been found from the starting node q_{init} to the goal node q_{goal}. BiRRTs keep the nice properties of RRTs such as probabilistic completeness [6].

The Connect Heuristic `CONNECT()` is a greedy function that can be considered an alternative to the `EXTEND()` function of RRTs described in algorithm 5 . It works similarly to the `CHECK_FINISHED()` function mentioned in Figure 3.4.6, but instead of greedily trying to reach q_{goal} after every q_{new} is created, it tries to reach the new node q_{new} from the closest node of the opposite tree by continuously iterating with step size ε towards q_{new}, until it either reaches an invalid state or connects. `CONNECT_BIRRT()` is shown in algorithm 6.

Function `BUILD_BIRRT()` is responsible for building both RRTs τ_I and τ_G and is shown in algorithm 7. First, it initialises both trees with their respective first nodes. Then, it

iteratively expands each RRT in succession with q_{new} using the function `EXTEND_BIRRT()` shown in algorithm 8. Note that `EXTEND_BIRRT()` is very similar to the `EXTEND()` function of RRTs. It does the same, except for skipping trying to connect each new node q_{new} to q_{goal}, as in this case the goal is to connect both RRTs. Therefore, `BUILD_BIRRT()` first defines q_{goal} after expansion to be the closest node to q_{new} of the opposite tree, which it then tries to greedily connect with using `CONNECT_BIRRT()`. If connection is successful, the path that links both RRTs is the solution and can be returned. If the connection fails, the iteration continues on the opposite tree, then back on the initial tree, etc. until either a connection is found and the solution returned or the maximum number of nodes I_{max} is reached and the problem deemed insolvable.

Algorithm 6: EXTEND_BIRRT(): Extend τ towards q_{rand} with distance ε. Note that this is very similar to EXTEND(), but lacks the q_{goal} check as q_{goal} changes as BUILD_BIRRT() progresses

Input: random node q_{rand}, an RRT tree τ, incremental distance ε
Output: Extend Status
begin
  $q_{near} \leftarrow NEAREST\_NEIGBOUR(q_{rand}, \tau)$;
  $q_{new} \leftarrow NEW\_CONFIG(q_{rand}, q_{near}, \varepsilon)$;
  if $q_{new} \neq NULL$ **then**
    τ.ADD_NODE(q_{new});
    τ.ADD_EDGE(q_{near}, q_{new});
    if $q_{new} = q_{rand}$ **then**
      // reached q_{rand}
      return *Reached*;
    else
      // extended towards q_{rand}
      return *Advanced*;
  else
    // $q_{new} \notin \mathcal{C}_{free}$
    return *Trapped*;
end

Algorithm 7: CONNECT_BIRRT(): Extend τ as far as possible towards q in a greedy fashion

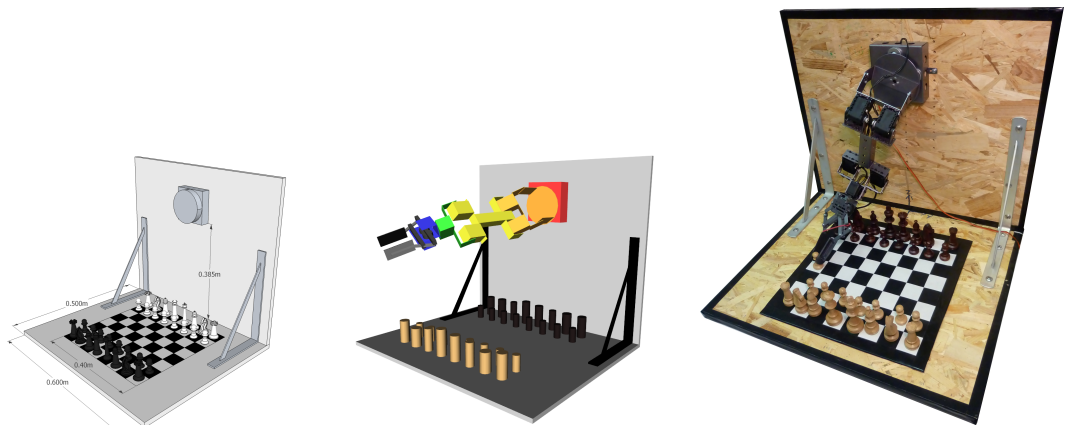
Input: Node q, RRT tree τ, incremental distance ε
Output: Status of connection attempt; either *Reached* or *Trapped*
begin
  repeat
    `// Incrementally step towards` q
    $S \leftarrow EXTEND\_BIRRT(\tau, q, \varepsilon)$;
  until $S \neq Advanced$;
  `//` q `is either` *Reached* `or` *Trapped*
  return S;
end

Algorithm 8: BUILD_BIRRT(): Construct two RRTs from q_{init} and q_{goal} respectively, that eventually connect thereby creating path between q_{init} and q_{goal} if one exists

Input: Initial configuration q_{init}, goal configuration q_{goal}, number of nodes I_{max}, incremental distance ε
Output: Path $q_{init}, \ldots, q_{goal} \subseteq \tau_I \cup \tau_G$
begin
  τ_I.INIT(q_{init}) ;
  τ_G.INIT(q_{goal}) ;
  for $i \leftarrow 1$ **to** I_{max} **do**
    $q_{rand} \leftarrow$ BIASED_RANDOM_CONFIG() ;
    if *EXTEND_BIRRT*(τ_I, q_{rand}, ε) $\neq$ *Trapped* **then**
      if *CONNECT_BIRRT*(q_{new}, τ_G, ε) = *Reached* **then**
        `// Solution found as` τ_I `and` τ_G `connected`
        return PATH(τ_I, τ_G);
    `// Continue expanding other tree`
    SWAP(τ_I, τ_G);
  `// No solution found within` I_{max} `nodes`
  return No Solution;
end

3.4.7 The Environment

Overview Before we look at the details of the manipulator hardware and how to interface with it, we focus on the environment. The environment consists of the mount, the chess board and the chess pieces.



(a) Proposed Environment **(b)** Simulated Environment **(c)** Real Environment

Figure 3.16: Comparison of environment representations

The Mount

The manipulator will be mounted on a vertical wall at a height of 370mm, such that it can reach the entire playing space, a subsection of the horizontal floor to which the vertical wall will be mounted using triangular brackets. The setup is robust and does not vibrate with the arm movement, therefore providing a stable environment for the chess pieces and eliminating one factor of accuracy decline. Figure 3.16 shows a comparison between the initially proposed environment, the simulated environment and a photo of the real result.

The wall and floor are constructed of 15mm thick pressed wood, each slab being 500mm*600mm in dimension. The triangular brackets constrain the arm's movement only at the extremities of its reach, where it does not affect the reachability in the chess playing zone and are large enough to provide the required robustness. The two wooden slabs form a 90° angle along their long sides. The camera is positioned above this setup in a non-fixed manner, facing downwards.

A plot of the manipulator's reachability is given in Table 3.1 that helped choose the manipulator height.

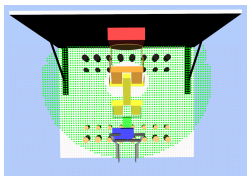
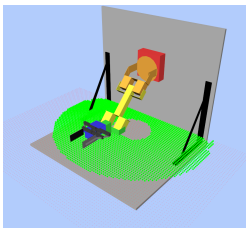
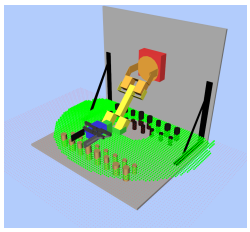
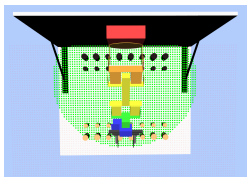
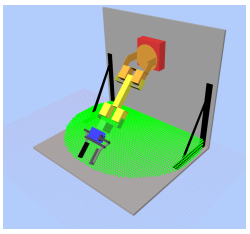
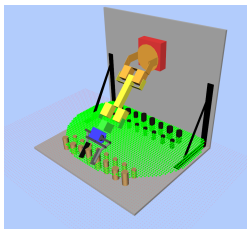
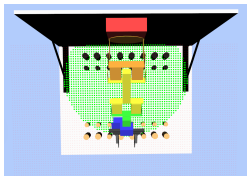
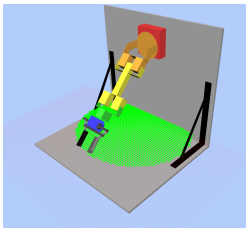
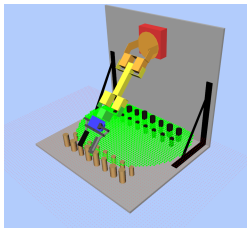
Base Height	Top View	3/4 View	3/4 Chess Pieces
height300mm			
370mm			
400mm			

Table 3.1: reachability for manipulators at different height attempting to reach points ${}^{goal}_{ref}\vec{P} = (i * 10mm\, j * 10mm\, 25mm)^T$. Note the unreachable zone directly under the arm for the 300mm height configuration, and the lack of corner reachability for the 400mm configuration

The Chess Elements

The Chess Board A usual 8x8 cell black and white chess board with the dimensions of 400mm*400mm*1mm board is used. The board is fixed and aligned to the floor board and the center is at ${}^{board}_{ref}\vec{P} = (450mm\, 0mm\, 0mm)^T$. This placement proved to have the best overall arm reachability while still leaving enough space around the edges. The board does not have any special markers, magnets, pressure pads or any other type of sensor or sensor aid, i.e. it is perfectly regular. Table 3.2 plots all the IK solutions for arm height 370mm for the center of every chess square at 25mm height. Generally, the more IK solutions per square center the better, as this gives the manipulator more approach angles which can be exploited later. Table 3.2 shows that the areas with most IK solutions are in the center two thirds of the board (especially rows 3 to 6), suggesting that a smaller board scaled to rows 3-6 would be better in terms of diverse reachability. However, that felt too inaccurate and clumsy to deal with such small pieces and movements. Therefore, the largest board possible was chosen according to Table 3.1.

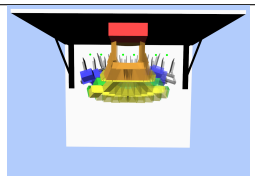
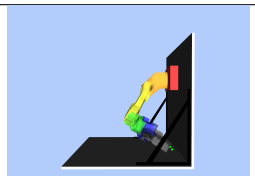
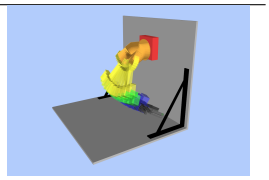
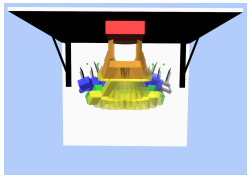
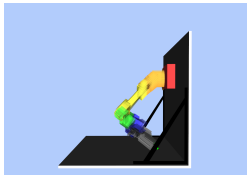
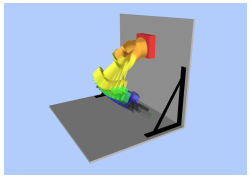
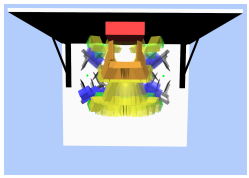
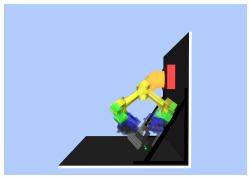
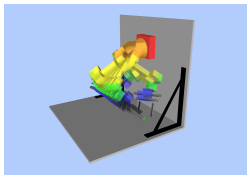
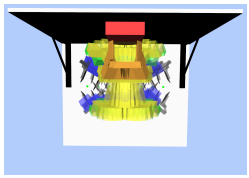
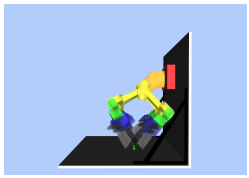
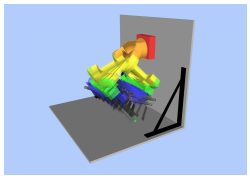
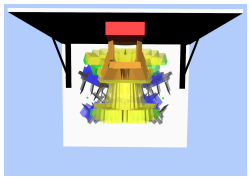
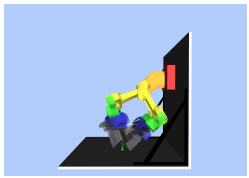
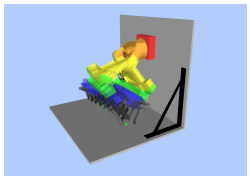
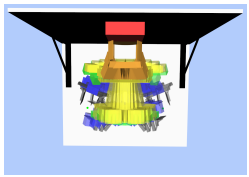
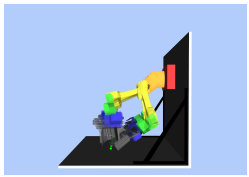
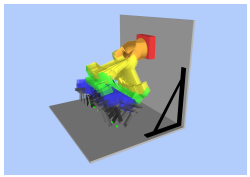
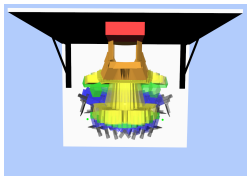
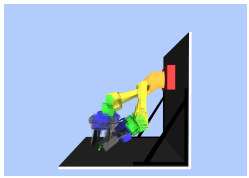
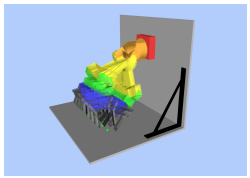
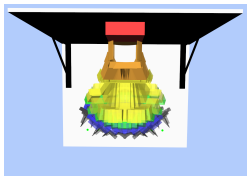
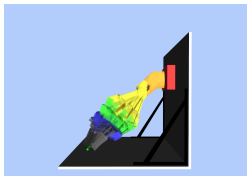
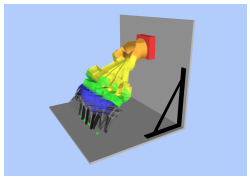
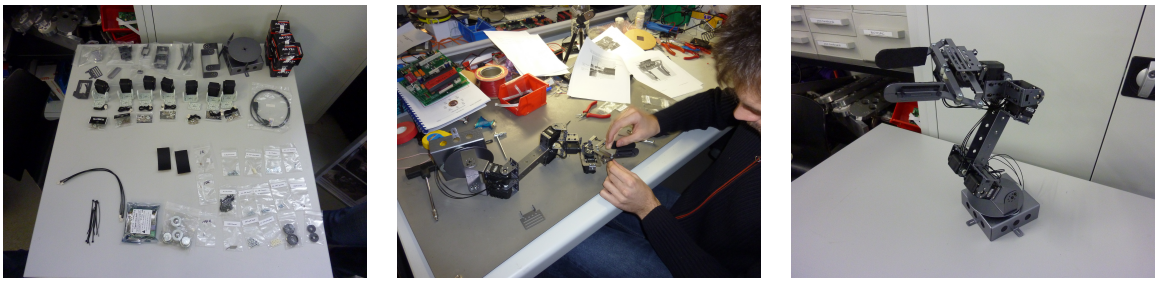
Row	Top View	Side View	3/4 View
0			
1			
2			
3			
4			
5			
6			
7			

Table 3.2: All IK solutions (with fixed wrist rotation) row-wise for the center of each chess square at 25mm height

The Chess Pieces Regular pieces are to used, more specifically the standardized Staunton 4 set (height of king: 78mm, base width 32mm). The pieces are made of wood and are matt black and beige. Initially we opted for larger pieces, but the board became too cluttered so we opted for the size mentioned above.

3.4.8 The Manipulator

Overview In this section we will focus on the actual hardware. To manipulate the chess figures an AX-12+ Smart Robot Arm by CrustCrawler was chosen due to its low cost, servo feedback and simplicity. It has four rotational degrees of freedom (4 DOF) and a single linear DOF gripper. The objective of the arm is to continuously send feedback and upon request, move successfully and accurately to a specified joint configuration, which it should be able to maintain. Figure 3.17 shows in the arm in pieces, in construction and fully assembled. The arm was built twice, once as a proof of concept to make sure all pieces were there, no defects present and experiment with different configurations. At first, we did not have the -45° pitch between DOF 0 and 1 as shown in the transformation in line 2 of Table 3.3. However, we intuitively decided that it would make sense to add this slight rotational offset to increase the IK solutions for certain goals. This can be observed in image (c) of Figure 3.11: if the rotation after the torso and before the shoulder would not exisit, the shoulder joint would like on the rotational axis of the torso, thereby potentially halving the available IK solutions. The two left solutions of (c) in Figure 3.11 would merge. Once we were satisfied with the (kinematic) structure of the arm, we rebuilt it and applied generous amounts of loctite to the nuts and bolts, as they were already loosening with just a few preliminary tests.



(a) Disassembled **(b)** Assembly **(c)** Assembled

Figure 3.17: Construction of the arm

Structure and Dimensions

Structure The reach radius around the manipulators' torso's (DOF 0) XY plane is 445mm. Due to this limited reach, the arm would not be able to reach across the chess

board. Therefore, the arm shown in Figure 3.18 with the CrustCrawler [15] setup will be mounted horizontally, i.e. rotated 90° around the Y axis so it sits on the global YZ plane resulting in a setup shown in Figure 3.16. The arm has a configuration specified in Table 3.3 with DOF in the order of kinematic chain. A corresponding image is shown in Figure 3.18.

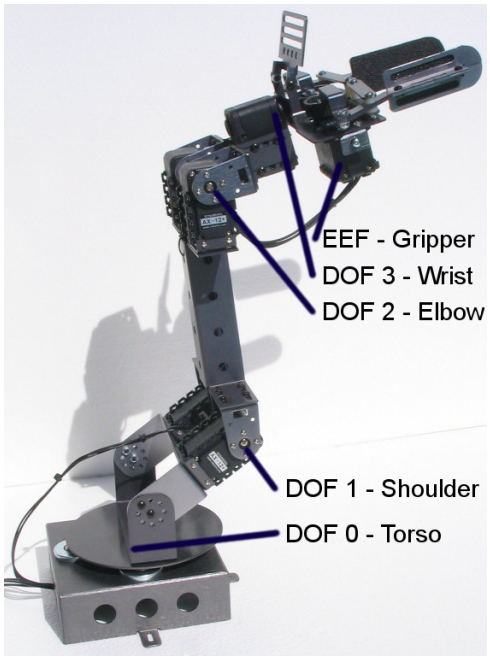


Figure 3.18: CrustCrawler AX12 Smart Arm with labelled DOF

Actuators To power the arm, a total of seven servos are used, where the mass bearing joints, i.e. the shoulder and elbow, use pairwise opposed servos. However, in our chosen configuration (see Section 3.4.7, the shoulder rotation is the most mass-bearing joint and unfortunately powered by a single servo only. This led to the servo shutting off if the arm's torso was rotated over ± 55° angle. The servos are Dynamixel AX-12 actuators, pictured in Figure 3.19. While not being the most powerful or accurate servos available, they are cost efficient, flexible and compact. They communicate through 8 bit half duplex asynchronous serial communication at a baud rate of 1000000 BPS. The actuators provide position, load, voltage and temperature feedback. To receive servo feedback and send commands, we will interface the servos to our host system with a USB2Dynamixel. Physically they link in a daisy chain fashion, i.e. all servos are connected in series forming a Dynamixel Network. Instruction packets are sent to the Dynamixel network that set registers in the servos, and status packets are returned exposing the content of the registers.

DOF	Joint	Servos	Description	Limits	Transform
0	Torso	Single	Rotation around Z axis (Yaw)	[-150°, 150°]	${}^{DOF0}_{base}T = \begin{pmatrix} 1 & 0 & 0 & 25 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 7.5 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
1	Shoulder	Double	Rotation around Y axis (Pitch)	[-120°, 120°]	${}^{DOF1}_{DOF0}T = \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 56.3 \\ 0 & 1 & 0 & 0 \\ \frac{-1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 100.7 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
2	Elbow	Double	Rotation around Y axis (Pitch)	[-120°, 120°]	${}^{DOF2}_{DOF1}T = \begin{pmatrix} 1 & 0 & 0 & 181.5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
3	Wrist	Single	Rotation around X axis (Roll)	[-150°, 150°]	${}^{DOF3}_{DOF2}T = \begin{pmatrix} 1 & 0 & 0 & 70 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 23 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
EEF(4)	Gripper	Single	Rotation converted into two linear actuators on XY plane	[-5°, 65°] [0mm, 40mm]	${}^{EEF}_{DOF3}T = \begin{pmatrix} 1 & 104.5 & 0 & 0 \\ 0 & 1 & 0 & \pm 31.6 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Table 3.3: Robot Arm Structure. Note *base* refers to the center and bottom of the manipulator base.

Servos are addressed by unique IDs. For a detailed description of the 49 RAM/EEPROM registers (i.e. temperature, goal position, etc) and instruction/status packet formats, see the AX12 Manual.

Initially we had some problems with the servos, so a tool with a GUI front-end was developed that directly interfaced to the Dynamixel network and could read/write to the registers. This debugging tool proved to be useful and the problems were discovered, understood and eliminated. The main problem seemed to be when the power supply could not keep up and servo IDs started to change causing conflicts on the Dynamixel network. If IDs are not unique, the whole Dynamixel network fails and one must manually check each individual servo one at a time and correct the IDs. Power supply issues were fixed by changing from a battery to a desk mounted dedicated power supply that could supply the required voltage and amps indefinitely. The exact inner workings of the diagnostic tool are not scope of this thesis. For details on the implementation of the interface used to control the arm/servos, see Section 3.4.9, the arm driver. For the basic features and specifications of the servos themselves, see Table 3.4.

Property	AX-12
Weight	55g
Gear Reduction	1/254
Input Voltage	7-10V
Max Holding Torque	12-16.5 kgf cm
Operating Angle	300°
Resolution	0.35°
Max Speed	114rpm

Table 3.4: Servo specifications

Figure 3.19: Dynamixel AX-12 Servo

3.4.9 SAC - Smart Arm Controller Node

Overview The Smart Arm Controller Node, or SAC for short, has the responsibility to bind the arm to the ROS network, thereby allowing ROS nodes to monitor the state of the manipulator and issue commands to it. We used the *ua-ros-pkg* developed by the Cognitive Robotics Group at the University of Arizona. Among other things, *ua-ros-pkg* provides a ROS interface to the Dynamixel network and an additional layer of abstraction to the servos.

Implementation Of the *ua-ros-pkg*, the smart_arm_controller node is of primary use to us. It loads parameters from a .yaml file that specify the properties of the joints we wish to control (such as joint limits, names, center positions, servo IDs) and then starts a controller spawner from ax12_controller_core that takes a list of joints we wish to instantiate that correspond to the previously loaded properties. In addition, a low-level serial driver is started which takes care of communicating with the Dynamixel network. All the relevant parameters such as baud rate, update rate, port name, etc. are specified in its own launch file. All of this is done through a single launch file. The launch files and configuration.yaml files were modified to fit our needs. This package is stable and flexible, but is not yet fully implemented. However, all the relevant features to enable motion planning are present. For the features that were missing (such as setting goal servo speeds), the diagnostic tool mentioned in Section 3.4.8 was used to write to the EEPROM registers which hold their value after shut down.

ROS Interface The smart_arm_controller node allows us to treat the Dynamixel network of servos in serial as a manipulator, thanks to the abstraction it provides, such as coupling opposed servos into a single joints and converting radians into 10bit integers (and back) that the AX12 servos use. It provides the ROS network with

1. the ability to set the goal angle of each joint
2. servo feedback which consists of: current position, load, temperature and voltage
3. joint feedback which consists of: position, goal position, error between goal and current position and velocity

Each joint subscribes to its own command message and publishes its own feedback. Servo feedback is provided as a published array of servo info at the update rate specified in the .yaml file mentioned above.

Note that the MCN communicates with the SAC, but with a control layer in between, outlined in Section 3.4.11

3.4.10 ORN - The OpenRave Node

Overview This section will explain the inner workings of the *The OpenRave Node* (*ORN*), the interface and functionality it provides to the ROS network and how this functionality is implemented. The ORN relies heavily on the OpenRAVE library mentioned in Section 3.4.4. We first define the functionality it provides and then look at these in greater detail. It is assumed that the real world can accurately be represented within the ORN, so that we can therefore plan in the simulation and apply the results to the real world.

Node Objectives and Capabilities

In short, this nodes provides the higher level planning algorithms that the MCN relies upon to move chess pieces. It sits as a lookup server on the ROS network where it can be asked to generate plans, visualise solutions and simulate movements. Together with the OpenRAVE library (see Section 3.4.4) it provides the following functionality either through a GUI or over the ROS network:

1. Initialise OpenRAVE with an accurate description of the environment and robot
2. Interact with the simulation - move the simulated arm joints and objects in the environment
3. Allow the state of the arm in the simulation to be polled
4. Change planners on the fly
5. Grasp and release simulated objects
6. Visualise and simulate the arm and environment
7. Visualise trajectories and multiple IK solutions
8. Visualise joint configurations and translations
9. Generate visual reachability maps
10. Filter IK solutions by various filters
11. Sort IK solutions by various preferences
12. Find the best IK solution to grasp from by minimising an error function
13. Find the best IK solution to release from by minimising an error function
14. Find the best pair of IK solutions to grasp then release from by minimising an error function

15. Generate a trajectory that brings the arm to a desired joint configuration
16. Generate a trajectory that brings the EEF or attached object to a desired translation
17. Generate a set of trajectories that moves a chess piece from a to b and the arm back to an initial configuration

A detailed explanation of the most important aspects in the list is given in the next few subsections.

Simulation and Visualisation

Concept The main concept behind the ORN is simulate the real world as closely as possible, then plan, calculate and solve with this simulated, virtual world, and apply the solutions to the real world. To do this, we use the OpenRAVE's viewer to visualise the state of the simulation, which comprises of the manipulator itself and the environment described in Section 3.4.7. The viewer also allows for interaction, allowing the user to move pieces and rotate arm joints in the simulated world. This is especially useful to create situations to test the arm, IK and planners with, to manually match the real world to the simulated world, to quickly run different experiments with different configurations and to intuitively tele-operate the arm. See Figure 3.16 for a visual comparison between the simulated world and real world.

Environment Description The ORN starts by initialising OpenRAVE and loading the environment. The environment is described in an .XML file that contains a list of kinematic bodies and their transforms. Simple shapes such as the floor and wall boards are represented as OpenRAVE boxes while more complex shapes such as the brackets that hold the boards together were modelled and are included as .iv files. A distinction is made between *static* bodies (i.e. the floor, wall, etc.) which cannot be moved and *dynamic* bodies (i.e. chess pieces) which can be manipulated. The .XML file is created in such a way that the resulting world frame results in the reference frame ref described in Section 3.4.1.

Chess pieces were initially modelled accurately. This causes problems as loading an environment with 32 highly detailed chess figures degraded performance so badly that the node could not be used. Therefore cylinders were used to approximate chess pieces. This was also problematic at first because the main collision checker (ODE) does not support cylinder $\leftrightarrow$ cylinder collision checks, allowing the simulation to not detect approximated chess pieces passing through one another. This was patched by a later ROS version of OpenRAVE, but we also approximated chess pieces with boxes.

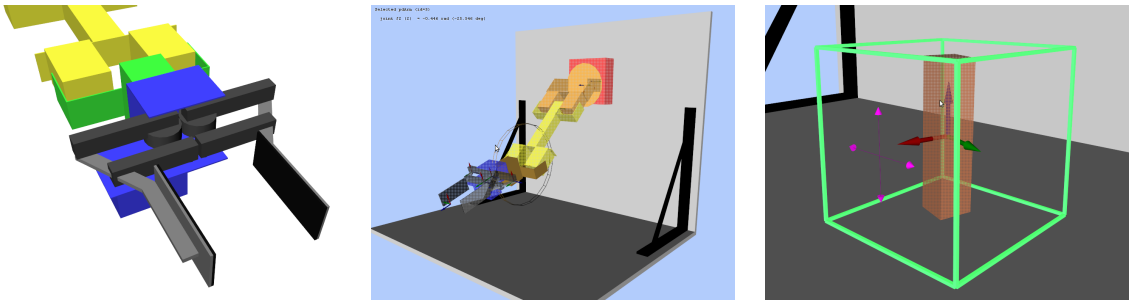
Robot Description Once the environment has successfully been loaded the robot is loaded. The robot is also described in the OpenRAVE XML robot format, which consists

of a kinematic chain description and EEF description. The robot file used was heavily inspired by a similar robot file from the Prairie Dog project of the Correll Lab of the University of Colorado, especially the kinematic XML description of the gripper. As the arm had a different configuration/dimensions a new description was made that modelled our manipulator and EEF more accurately.

Once the robot has loaded, the robot specific IK solver and planner are loaded.

Pitfalls The obvious pitfall is the assumption that the real world we act in and the simulated world we plan in are the same. Tiny errors in rotation or dimension can have large consequences. The biggest problem is accurately simulating object $\leftrightarrow$ EEF interactions and gravity, which can cause the real world manipulator to droop. For more specific information, please see autorefssec:manexp

A couple of OpenRAVE viewer screenshots are shown below in Figure 3.20.



(a) Simulated EEF **(b)** Rotation a joint **(c)** Moving a kinbody

Figure 3.20: Three screen shots showing details of the OpenRAVE simulation and interaction

Move to Joint Configuration

Functionality Given an initial joint configuration, find a trajectory that reaches a given joint configuration.

Details This functionality is provided as a service over the ROS network. The initial joint configuration is assumed to be the current joint configuration of the simulated arm, which can be set at any time. The function is called with a goal configuration and the ORN simply uses the currently selected planner to generate a trajectory, which is then parsed to a simplified format containing nothing more than a list of joint configurations which it then returns, along with the total number of configurations. If no such trajectory is found (none exists or the start/goal configuration is invalid) a negative total is returned, thus indicating failure.

Move to ${}^{goal}_{ref}P$

Functionality Given a goal transform, find a trajectory that places the EEF (or optionally grasped object) on it. This is similar to Section 3.4.10 (it actually calls it) but with an additional step: Before it knows where to must find an IK solution that it can move to. Once an IK solution is found, it can call *Move to Joint Configuration*. If this fails, it can optionally retry with a different IK solution where the first 3 DOFs differ.

This function is actually the main work horse of the ORN and can be optimised in many ways. As Section 3.4.5 explains and both Figure 3.11 and Table 3.2 show, many IK solutions may exist, so naturally the question arises: Which IK solution do we pick as the goal configuration? The short answer: It depends.

Simple Case - Speed If speed is a goal, we simply take the first IK solution the IK solver finds. This is useful for two things:

Quickly testing if a solution exists Sometimes this is desirable. For example, the MCN graphically display in real time if a solution exists by displaying a small green point instead of a red one.

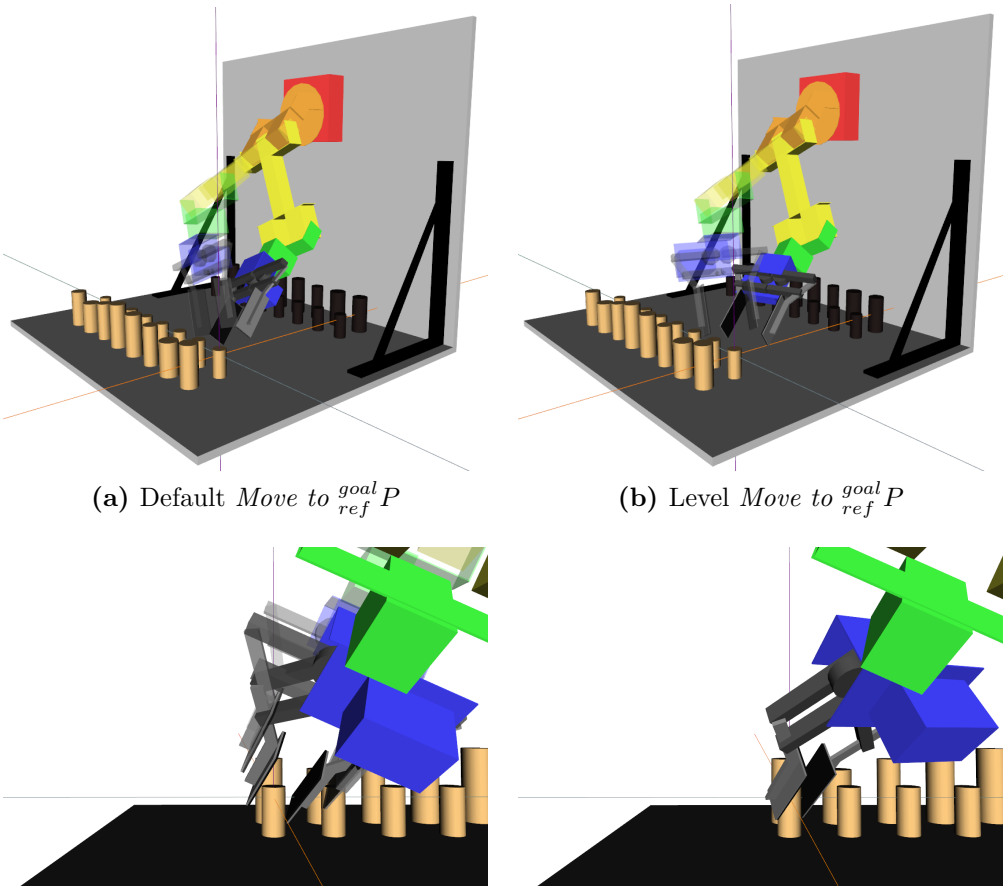
Grasping objects that just fit inside the gipper In this case, the found IK solution is probably unique and/or very good. If the object is large enough the gripper will have to come from a certain angle to accommodate it. Normally the angle is very well aligned. This generally works very well in simulation cannot really be applied to real world situations.

Quickly moving to an unimportant position If one does not care about the end configuration and just wishes to place the EEF in a different zone, this approach is the fastest when moving to a goal translation. Generally though, one would use *Move to Joint Configuration* in these situations.

Optimise Grasp - Levelness When moving towards an object to grasp it, it is desirable to find a solution with an angle of approach that maximises the chance of getting a good grip on the object when a grasp is attempted. With our simple gripper and restricted DOF (and therefore limited angles of approach) the most successful solution was generally the most level one (level is defined in Section 3.4.3). This is more of a real world optimisation rather than a simulation based one. The simulator does not care at what angle the object is grasped: it closes the grippers until a contact between the object to grasp and the gripper is identified, at which point the gripper holds its position and the object is *attached* to the kinematic chain of the arm and considered grasped. However, in the real world the gripper would move the object in a non-determinable way before (hopefully) grasping it. Using a level approach helps two-fold: (1) It increases the likelihood of a successful real world grasp and (2) it helps keep the real world and simulation

world synced, as a level grasp is usual determinable (i.e. the object behaves as expected while being grasped.

So, when moving towards a translation to grasp an object one first calculates all the IK solutions, and then rates each solution with $levelness(^{EEF}_{ref}T)$, defined in (3.22). Normally there are two "perfectly level" solutions for every unique IK solution of the first 3 DOF: one with the wrist upside down and one with the wrist upright. At this state we do not have preference towards either, as we do not know where we are moving the object to next. See Section 3.4.10 for the case where we do know in advance where the object is going to be released.



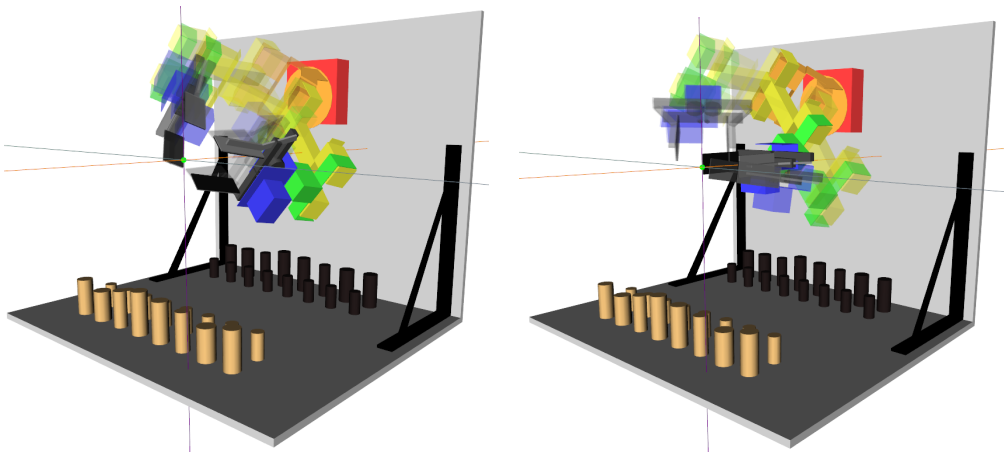
(a) Default *Move to* $^{goal}_{ref}P$ **(b)** Level *Move to* $^{goal}_{ref}P$

(c) Default Grasp **(d)** Level Grasp

Figure 3.21: Ik solutions applied to a grasp task. Top row shows the resulting configuration of *Move to* $^{goal}_{ref}P$ and the bottom row a zoom in of the resulting grasp. Imagine what the grasp and grasped object would look like in real life, and in which case the simulation would be most authentic

In summary, grasping an object with a level EEF helps real life grasps be more suc-

cessful and the simulation simulate the interactions more accurately. Figure 3.21 shows two IK solutions applied to a grasping task: the first found solution and the most level solution. Looking at how the post grasp looks, one might find it intuitive that a level grasp is more likely to succeed. Figure 3.22 shows IK solutions, pairwise level and non level.



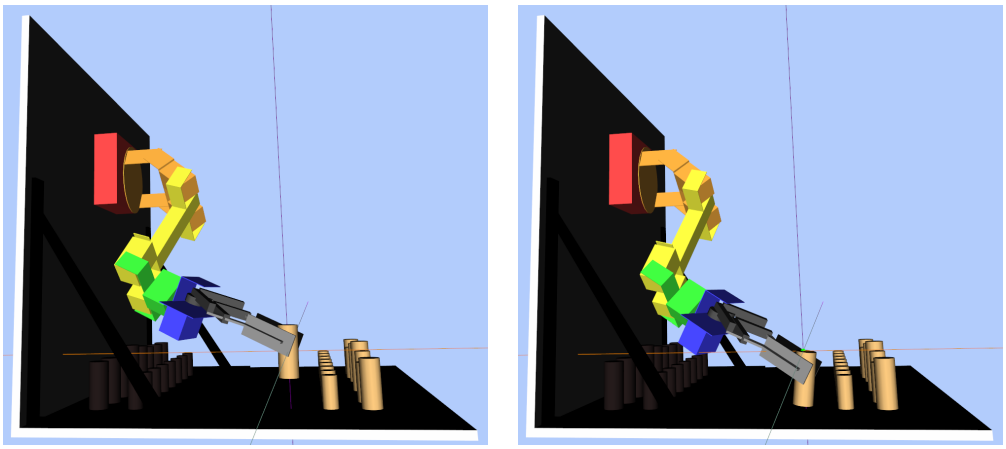
(a) First solution **(b)** Level solution

Figure 3.22: Showing all IK solutions (with unique first 3 DOF). Left shows the first found solution and right the most level of all found solutions.

Optimise Release - Uprightness EEF levelness also helps when releasing objects for the same reasons as it helps with grasping. However, we can improve on this as Figure 3.24 shows. When we placing a wine glass down on a table, we try to orientate it such that the bottom of the glass is parallel to the table top to minimise the chance of the glass falling over. The same idea can be applied to our manipulator. In Section 3.4.3 we devised the function (3.23) which given a transform T returns the degrees away from perfect uprightness. To obtain the best joint configuration that places the object at a given translation ${}^{goal}_{ref}P$, we commence as follows:

- Determine the transform ${}^{obj}_{eef}T$ between the grasped objected obj and the EEF eef as explained in Eq. 3.20 and in Section 3.4.5 (See algorithm 9 `CALC_OFFSET()`)
- Extract the translation ${}^{obj}_{eef}P$ component from transform ${}^{obj}_{eef}T$
- Optional: Calculate the maximum height of the object and update goal translation. To do this, we create a bounding box around the object and compute the largest (diagonal) distance from one corner to the opposite. This gives us the worst case scenario height of the object if placed on its tip. The height is therefore half of this and can replace the z component of the goal translation ${}^{goal}_{ref}P$. See algorithm 11 `CALC_MAX_HEIGHT()`

- Call the TranslationGlobalLocal6D IK solver with goal=${}^{goal}_{ref}P$ and offset=${}^{obj}_{eef}T$ to obtain all IK solutions. Explained in Section 3.4.5
- Evaluate every solution with `uprightness()`: Set joints to solution values, get new ${}^{eef}_{ref}T$, multiply with ${}^{obj}_{eef}T$ (See (3.19)), feed into `uprightness()` to get rotation offset from perfect uprightness
- Pick solution with least rotation offset (See algorithm 12 `GENERATE_PQ()`)
- Call planner to reach picked solution
- Optional: if failed, dismiss any other solutions that one can reach from the one attempted and try with next best. Repeat until none left or solution found (See algorithm 13 `GET_BEST_TRAJ()`)

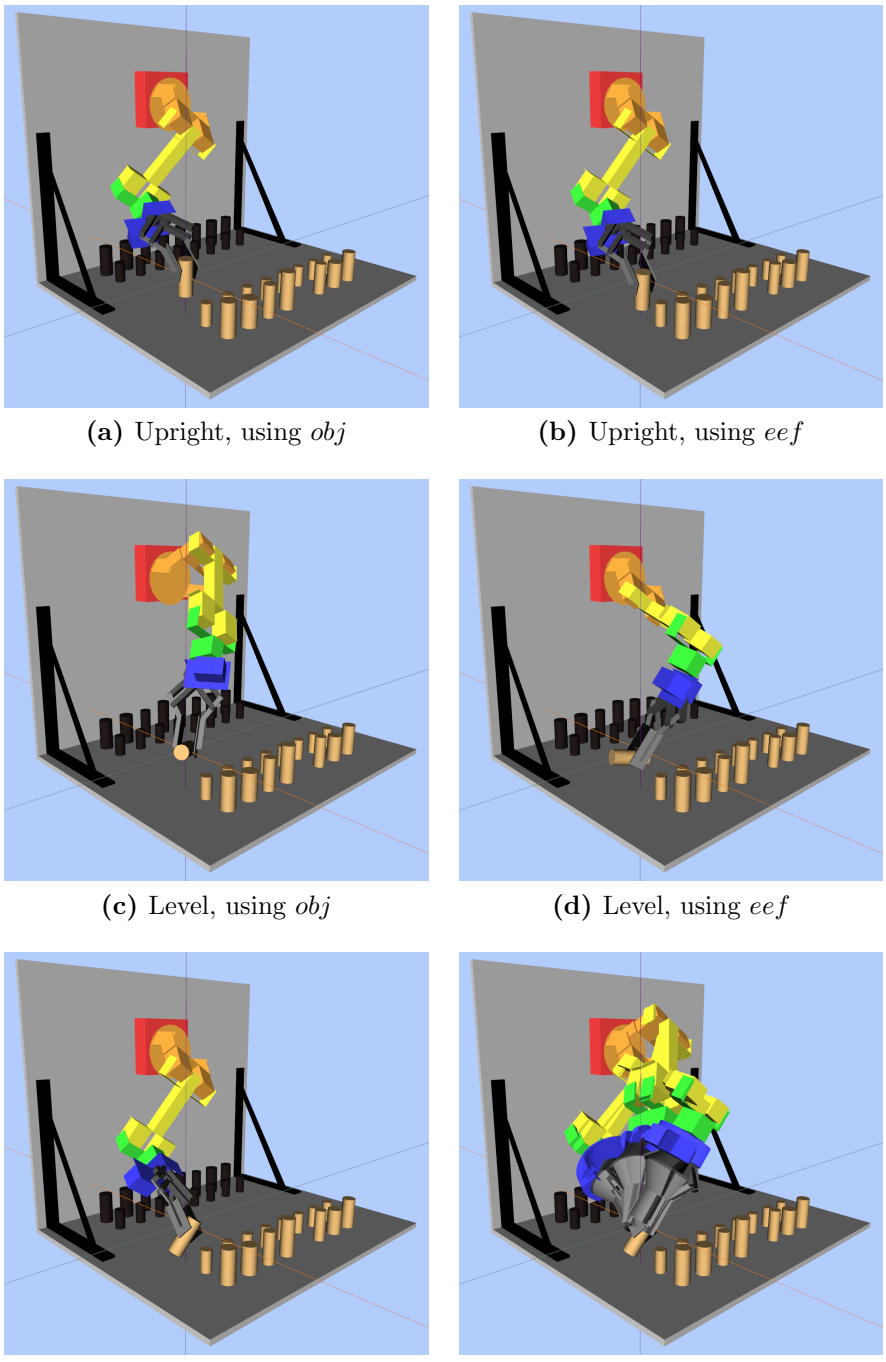


(a) Default: the object center is at the given translation

(b) `autoheight()` applied: the object center is as low as possible under the given translation

Figure 3.23: Demonstrating the function `autoheight()` that can be applied when working with the object as part of the kinematic chain

The images in Figure 3.24 show the benefits of working with the grasped object over the EEF and what advantages selecting an upright solution can have. Note that by using *obj* as reference we increase the number of IK solutions (intuitively, if the arm was less constrained an IK solution plot would form a sphere around the goal). Working with *obj* also allows us to apply `autoheight()` demonstrated in Figure 3.23. Using `autoheight()` has the further benefit that it helps keep objects as low as possible in the simulation without having to apply gravity (as this is still very much unsupported).



(a) Upright, using *obj*

(b) Upright, using *eef*

(c) Level, using *obj*

(d) Level, using *eef*

(e) Default, using *eef*

(f) All, using *obj*

Figure 3.24: Showing difference between using *eef* or *obj* as reference, as well as selecting level or upright solutions. Note that in this case the first two images are almost the same - in this case it is obvious why, but this is not generally the case as often it is better to grasp from a higher angle than a lower one

Algorithm 10 `PLAN_TO()` compiles the ideas mentioned in this section into one universal function that generates a plan from the current configuration to a goal translation ${}^{goal}_{ref}P$. It takes either $ToGrasp$, $ToRelease$, $Fast$ as an argument to specify the ultimate objective of the plan. The following functions are assumed known:

GET_EEF(a) gets the active EEF of a
GET_GRASPED_OBJ(a) returns the object grasped by a if one exists, otherwise NULL
CALC_BOUNDING_BOX(a) calculates a bounding box of
GET_DIM(a) get the dimensions of a, returns vector with 3 elements representing the size in x,y,z direction
GET_TRANSFORM(a) returns the transform of a
TranslationGlobalLocal6D_IK(goal, offset, x) calls IK function described in Section 3.4.5 and returns the first X solutions
GET_BiRRT_PLAN(a,b) generates a trajectory from configuration a to configuration b using the BiRRT planner described in algorithm 7
PRIORITY_QUEUE.INIT() create an empty priority queue
INSERT_WITH_PRIORITY(element, pri) add element with priority pri to a priority queue
LEVELNESS(transform) function defined in Section 3.4.3
UPRIGHTNESS(transform) function defined in Section 3.4.3
SET_JOINT_CONFIG(config) set the simulated arm's joints to the configuration config

Algorithm 9: CALC_OFFSET(): Determine the translation from EEF eef to the object obj

Input: Grasped object obj, EEF eef)
Output: Translation ${}^{obj}_{eef}P$
begin
`// Get transforms`
${}^{eef}_{ref}T \leftarrow$ GET_TRANSFORM(eef);
${}^{obj}_{ref}T \leftarrow$ GET_TRANSFORM(obj);
`// Determine transform from EEF to object`
${}^{obj}_{eef}T \leftarrow {}^{eef}_{ref}T^{T} \cdot {}^{obj}_{ref}T$;
`// Extract translation from transform`
${}^{obj}_{eef}P \leftarrow ({}^{obj}_{eef}T_{3,0}\ {}^{obj}_{eef}T_{3,1}\ {}^{obj}_{eef}T_{3,2})^{T}$;
return ${}^{obj}_{eef}P$;
end

Algorithm 10: PLAN_TO(): Create a trajectory that brings the specified manipulator's EEF eef or grasped object obj to goal translation ${}^{goal}_{ref}P$ such that the final joint configuration is optimised for objective O. If an obj is grasped and A is $True$, put the object on the ground

Input: Goal translation ${}^{goal}_{ref}P$, objective O, autoheight A, manipulator arm
Output: Trajectory $traj$
begin
  $eef \leftarrow GET\_EEF$(arm) obj$\leftarrow GET_GRASPED_OBJ$(eef) ${}^{obj}_{eef}P \leftarrow (0 \quad 0 \quad 0)^T$; **end**
  if $obj \neq NULL$ **then**
    `// An object is attached`
    if $A = True$ **then**
      `// Calculate max height of` obj `and update` ${}^{goal}_{ref}P$
      ${}^{goal}_{ref}P_z \leftarrow$ CALC_MAX_HEIGHT(obj);
    `// Get translation between grasped object and EEF`
    ${}^{obj}_{eef}P \leftarrow$ CALC_OFFSET(obj, eef)
  if $O = Fast$ **then**
    `// Determine first solution quickly`
    $jointConfig \leftarrow$ TranslationGlobalLocal6D_IK(${}^{goal}_{ref}P$, $offset$, 1);
    if $jointConfig \neq NULL$ **then**
      $traj \leftarrow$ GET_BiRRT_PLAN(arm.CONFIG(), $jointConfig$);
      if $traj \neq NULL$ **then**
        `// Path to` $jointConfig$ `found and trajectory generated`
        return $traj$;
    return $FAILED$;
  else
    `// Optimising for` $Release$ `or` $Grasp$ `objective`
    $jointConfigList \leftarrow$ TranslationGlobalLocal6D_IK(${}^{goal}_{ref}P$, ${}^{obj}_{eef}P$, ALL);
    if $jointConfigList = EMPTY$ **then**
      return $FAILED$;
    else
      `// Sort IK solutions by objective value`
      $pq \leftarrow$ GENERATE_PQ(arm, O, $jointConfigList$);
      `// Return the trajectory to the best reachable IK solution`
      return GET_BEST_TRAJ(pq, arm);

Algorithm 11: CALC_MAX_HEIGHT(): Determine the maximum height of *obj* by assuming the worst case scenario: *obj* stands on one corner with its opposite corner (diagonally across) is as high as possible

Input: An object *obj*)
Output: Maximum height of object *obj*
begin
    `// Obtain objects bounding box dimensions`
    $bb \leftarrow CALC\_BOUNDING\_BOX$(obj) dim$\leftarrow$ GET_DIM(bb);
    `// Calculate diagonal of bounding box`
    $max\_height \leftarrow \frac{\sqrt{dim_x^2+dim_y^2+dim_z^2}}{2}$;
    return max_height;
end

Algorithm 12: GENERATE_PQ(): Create a queue of solutions *pq* from IK solutions $jointConfigList$ ranked by their value according to objective O

Input: manipulator arm, objective O, all IK solutions $jointConfigList$)
Output: Trajectory $traj$
begin
    $arrmconfig \leftarrow arm$.CONFIG() $eef \leftarrow$ GET_EEF(arm);
    $pq \leftarrow$ PRIORITY_QUEUE.INIT($\emptyset$);
    if $O = Grasp$ **then**
        `// Prioritise EEF levelness`
        for $config \in jointConfigList$ **do**
            arm.SET_JOINT_CONFIG($config$);
            ${}^{eef}_{ref}T^T \leftarrow$ GET_TRANSFORM(eef);
            pq.INSERT_WITH_PRIORITY($config$, LEVELNESS(${}^{temp}_{ref}T^T$));
    if $O = Release$ **then**
        for $config \in jointConfigList$ **do**
            `// Prioritise grasped object's uprightness`
            arm.SET_JOINT_CONFIG($config$);
            ${}^{eef}_{ref}T^T \leftarrow$ GET_TRANSFORM(eef);
            ${}^{temp}_{ref}T^T \leftarrow {}^{eef}_{ref}T^T \cdot {}^{obj}_{eef}T$;
            pq.INSERT_WITH_PRIORITY($config$, UPRIGHTNESS(${}^{temp}_{ref}T^T$));
    `// Return arm to initial configuration`
    arm.SET_JOINT_CONFIG($arrmconfig$);
    return pq;
end

Algorithm 13: GET_BEST_TRAJ(): Attempts to find a trajectory from the arm's current state to the best valid and reachable joint configuration. Notice the following optimization: Similar joint configurations are assumed to be similarly valid and assumed to have similar positions in the priority queue. Therefore instead of repeatedly trying to connect the current arm configuration to similar and probably invalid states, remove large invalid portions first: Once an invalid configuration $q_{invalid}$ is found, all the configurations $q \in pq$ that the planner can connect to $q_{invalid}$ are also removed from pq with the rational that if the planner could reach $q_{invalid}$, it could also reach q. BiRRT is really fast when planning over small distances, potentially saving much time by avoiding repeatedly planning over long distances

Input: Priority queue pq, manipulator arm
Output: Trajectory $traj$
begin
  while $pq \neq \emptyset$ **do**
    $config \leftarrow pq$.POP_HIGHEST_PRI_Element();
    $traj \leftarrow$ GET_BiRRT_PLAN(arm.CONFIG(), $jointConfig$);
    if $traj \neq NULL$ **then**
      `// Path found to best valid reachable configuration`
      return $traj$;
    else
      `// No path to current best configuration found`
      for $config \in pq$ **do**
        `// Remove all configurations reachable from latest non-reachable`
        if *GET_BiRRT_PLAN(arm.CONFIG(), jointConfig)* $\neq$ *NULL* **then**
          pq.REMOVE($config$);
  return FAILED;
end

Move Chess piece from ${}^{start}_{ref}P \rightarrow {}^{goal}_{ref}P$

Pre-computing Grasp and Release pair Due to the lack of DOF of the manipulator, it is very constrained in the number of approaches it has to grasp and release objects. While it can grasp a chess piece in every position perfectly levelly (look at Table 3.2 closely, every IK solutions is within 0.0001^c level), it often cannot release chess pieces within 45° of perfect uprightness. Therefore, we must make use of every possible opportunity to maximize the chance of being able to release a chess piece safely. If multiple level grasp angles exist, we must choose the one that allows us to eventually release the object as upright as possible.

Functionality Just as *Move to* ${}^{goal}_{ref}P$ (described in Section 3.4.10), *Move Chess piece from* ${}^{start}_{ref}P \rightarrow {}^{goal}_{ref}P$ can be called over the ROS network. It is the ultimate function combining all of the optimizations described so far and is used to move a chess piece from ${}^{start}_{ref}P \rightarrow {}^{goal}_{ref}P$ and optional move the manipulator away to a given configuration. Because this function can take 10s of seconds to complete, continuous status updates are returned to the caller such that it can follow the progress. It computes the total number of steps (which vary depending on factors such as the number if generated IK solutions) and the current step, allowing the caller to observe the progress continuously.

Details The function works as follows.

- First, it calculates every IK solution for the ${}^{start}_{ref}P$ and sorts by levelness, i.e. optimized grasp configurations.
- To reduce the search space later on, only the top 4 solutions are kept. This number makes sense as a maximum of 4 IK solution can exist where the first 3DOF are unique. Optionally, all IK solutions within a certain levelness threshold can be kept.
- For each IK solution kept, it checks if an object is within a close distance, and if so computes ${}^{obj}_{eef}T$. If not, it assumes ${}^{obj}_{eef}T$ to be I_4. ${}^{obj}_{eef}P$ is also extracted.
- Then, for each IK solution and corresponding ${}^{obj}_{eef}T$ and ${}^{obj}_{eef}P$, TranslationLocalGlobal6D is called with goal=${}^{goal}_{ref}P$ and offset=${}^{obj}_{eef}P$. Using ${}^{obj}_{eef}T$, it calculates ${}^{obj}_{ref}T$ which it can plug into `uprightness()` to evaluate the uprightness that this combination of start and goal joint configurations would make *obj* have.
- Each start and goal pair is saved and put into a priority queue with priority=uprightness
- We continuously pop the most prioritized element from the priority queue and plan to the popped start configuration from the arms initial state, then from the start to the goal configuration, and optionally back to a specified return configuration. If all these plans succeed, we stop popping and return the 3 resulting trajectories.

The resulting two/three trajectories would thereby contain the best combination of start and goal configurations that maximizes the chance to correctly grasp an object and

maximizes the chance of successfully releasing it.

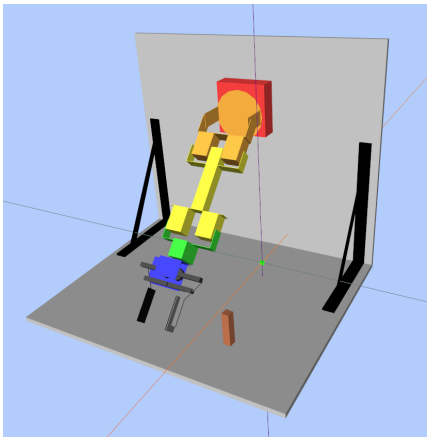
Figure 3.25 shows an object being moved from its initial position to the translation indicated by the green dot. The left sides shows the iterative approach (go to configuration resulting in most level EEF, grasp, go to configuration resulting in the most upright object transform, release) and the right side the pre-computed pair. In both cases, each grasp configuration was optimized for EEF levelness and each release pose for object uprightness. It might seem unintuitive that the release configuration of the sequential case really is the most upright possible, but it has no better option without first releasing the object elsewhere and grasping from a different angle. This is exactly what pre-computing avoids. Here it is clear that pre-computing pairs can greatly help. The pseudo-code for the algorithm is given below in algorithm 14.

Algorithm 14: GRASP_PLAN(): Precomputes best IK pairs to move an object from ${}^{start}_{ref}P$ to ${}^{goal}_{ref}P$ and optionally move the arm to $config_{final}$, return a trajectory for each interval

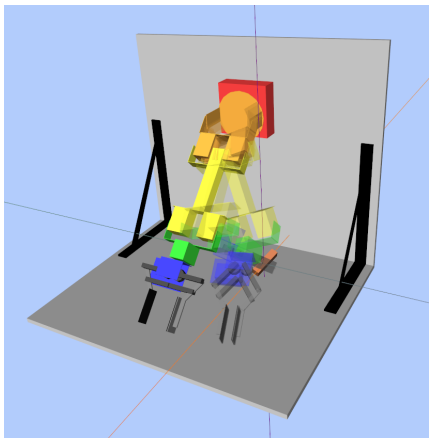
Input: Translation ${}^{start}_{ref}P$, translation ${}^{goal}_{ref}P$, joint configuration $config_{final}$, manipulator arm
Output: Trajectory $traj_{start}$, $traj_{goal}$ and optionally $traj_{away}$
begin
  $arrmconfig \leftarrow arm$.CONFIG() `// Get IK pairs sorted by goal object levelness`
  $pq \leftarrow$ GET_PAIR_PQ(${}^{start}_{ref}P$, ${}^{goal}_{ref}P$, arm) `// Attempt to plan between the pairs and optional` $config_{final}$
  for ($config_{start}$, $config_{start}$) $\in pq$ **do**
    if $traj_{start} \leftarrow$ *GET_BiRRT_PLAN(arrmconfig,* $config_{start}$*)* $\neq$ *NULL* **then**
      if $traj_{goal} \leftarrow$ *GET_BiRRT_PLAN(*$config_{start}$*,* $config_{goal}$*)* $\neq$ *NULL* **then**
        if $config_{final} \neq$ *NULL* **then**
          `// Planning to move away too`
          if $traj_{start} \leftarrow$ *GET_BiRRT_PLAN(*$config_{goal}$*,* $config_{final}$*)* $\neq$ *NULL* **then**
            `// No plan failed, return them`
            return $traj_{start}$, $traj_{goal}$, $traj_{away}$;
        else
          `// No plan failed, return them`
          return $traj_{start}$, $traj_{goal}$;
  `// Either could not find/plan between IK solutions`
  return FAILED;
end

Algorithm 15: GET_PAIR_PQ(): Generates IK pairs to move an object from ${}^{start}_{ref}P$ to ${}^{goal}_{ref}P$ and inserts them into a queue prioritised by grasped object uprightness. Note that GET_IK_PQ() is the same as PLAN_TO() but returns pq instead of $traj$.

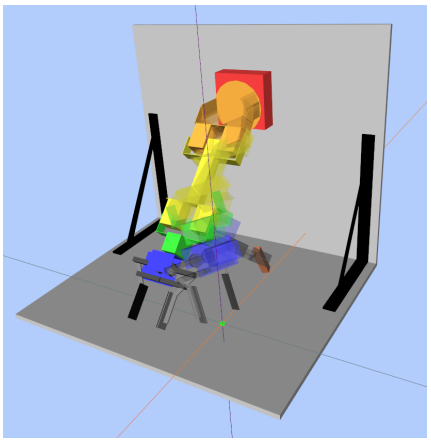
Input: Translation ${}^{start}_{ref}P$, translation ${}^{goal}_{ref}P$, manipulator arm
Output: Priority queue pq
begin
  $pq \leftarrow$ PRIORITY_QUEUE.INIT($\emptyset$);
  `// Get best grasp IK solutions`
  $pq_{start} \leftarrow$ GET_IK_PQ(${}^{start}_{ref}P$, $Grasp$, $False$, arm);
  if $pq_{start} = NULL$ **then**
    return NULL;
  else
    pq_{start}.SET_MAX_SIZE(4);
  for $config_{start} \in pq_{start}$ **do**
    arm.SET_JOINT_CONFIG($config_{start}$);
    arm.GRASP();
    $obj \leftarrow arm$.GET_GRASPED();
    ${}^{obj}_{ref}T \leftarrow obj$.GET_TRANSFORM();
    `// Get best release IK solutions`
    $pq_{goal} \leftarrow$ GET_IK_PQ(${}^{goal}_{ref}P$, $Release$, $True$, arm);
    if $pq_{goal} \neq NULL$**then**
      for $config_{goal} \in pq_{goal}$ **do**
        arm.SET_JOINT_CONFIG($config_{start}$*);*
        pq.INSERT_WITH_PRIORITY(($config_{start}$, $config_{goal}$*),*
        $config_{goal}$*.GET_PRIO());*
    `// Reset situation for next round`
    arm.RELEASE();
    obj.SET_TRANSFORM(${}^{obj}_{ref}T$*);*
  return pq;
end



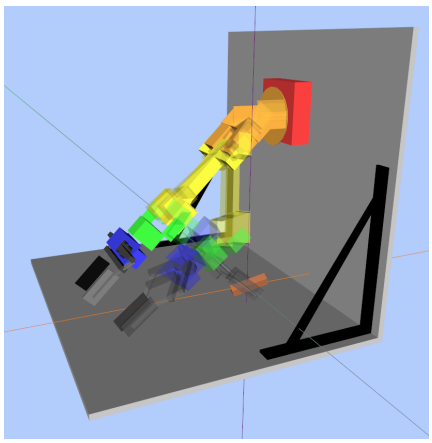
(a) Initial Configuration



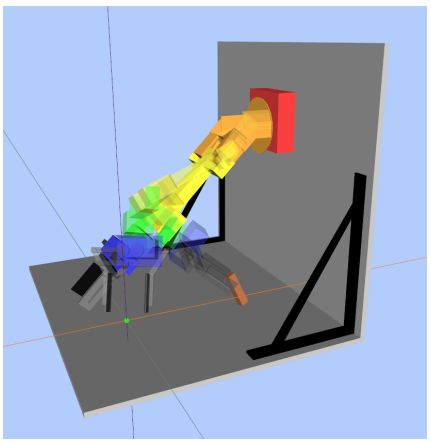
(b) sequential



(c) pre-computed



(d) sequential



(e) pre-computed

Figure 3.25: Demonstrating the benefits of pre-computing IK solution pairs. The chosen grasp and release IK solution are shown as transparent overlays.

3.4.11 MCN - The Manipulation Core Node

Overview The Manipulation Core Node, or MCN, essentially ties together the functionally of both the SAC (detailed in Section 3.4.9) and ORN (detailed in Section 3.4.10) together with a complex GUI that allows the user to visualize, intercept and set what is going on behind the scenes. All the functionality is exposed and all relevant parameters can be set. It is the client of the ORN and therefore works tightly with the visualization. As much as the MCN is very much user-interaction based, it can also work fully anonymously, accepting chess moves over the ROS network that the CCN issues. (See Section 3.2). While autonomously managing the hardware and ORN, it continues to provide visual feedback to the user. The MCN can also be used without the SAC, allowing the user to interactively query the ORN and visualize the results. Likewise, it can also be used without the ORN and allow the user to monitor and tele-operate the manipulator. If both the SAC and ORN are available, the user can also tele-operate the arm with assistance, however that is not scope of this thesis. The whole node was written in c++, using QT as the GUI framework and roscpp to communicate with the ROS network.

Structure and Graphical User Interface The GUI and program structure are divided into 3 main categories according to the level of abstraction: the low-level (i.e. servos), medium level (i.e. joints) and high level(i.e. manipulator.) Each level has its corresponding visualization and control. However the control at lowest level is not possible due to possibly damaging the servos (especially the opposed ones). In the following sections we will explorer each level of abstraction and the corresponding GUI elements.

Low-Level: Servo Interface

This level essentially deals with communicating with the SAC. It is subscribed to the SAC's servo messages that it receives at 10hz. It parses these and updates the GUI and a local buffer. This buffer is used in two ways: as a cache to read from the servos and as a buffer to write to them. For example, when a slider with 1000 increments that represents a servo's goal angle is moved from left to right, it is very easy to flood the Dynamixel network if one would send a new goal position every time a new increment is selected. Therefore, using the cache/buffer, the Dynamixel network is only addressed at most, once every 0.01ms and only if the servos goal angle register differs from the buffers goal within the servos resolution. This prevents flooding the network with similar messages that have no use and too many different messages at once. Figure 3.26 shows the servo tab displaying all the relevant information received from the SAC. Note that the min/max/init values do not dynamically update as they are received from the parameter server at start up and do not change.

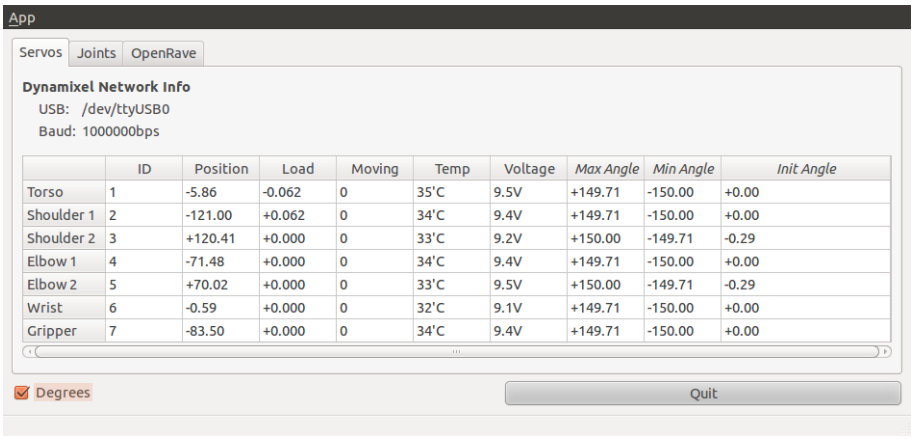


Figure 3.26: The individual servo details are updated at 10hz

Medium-Level: Joint Interface

This is very similar to the low-level, except for it provides the abstraction from servo to joint that the SAC provides. The user can directly control the position of joints and view their properties, as well as graph their goal and current positions. For more details see the corresponding images Figure 3.27, Figure 3.28 and Figure 3.29 below.

High-Level: OpenRAVE Interface

The high-level section deals with the ORN, asking it to generate plans that are requested by the user or as indirect consequences of the CCN requesting a chess move.

The user can select which portions of the chess piece moving tasks are done automatically (i.e. upon request via the ROS network) or where manually. The user can call all algorithms specified in Section 3.4.10 along with the respective parameters such as creating trajectories to move to joint configurations, given translations, or complete pickup and release type moves. Optimizations can be enabled or disabled to demonstrate their benefit.

In addition, the user can preview joint configurations and translations in real time. For example, sliders can be used to visually move/select a translation around. The translation is marked by a point that changes color in real time depending on kinematic reachability. While moving the translation around, the user can optionally view an IK solution in real-time or view all solutions (with a 1 second delay).

The manipulator can sync to the OpenRAVE simulated arm (i.e. moving the simulated arm moves the real arm) or vice versa (i.e. the simulated arm follows the joint configuration of the real arm in real-time). Generated trajectories can be previewed, executed, executed in reverse and the speed of execution can dynamically be changed, also in real-time during an execution. Manual grasping, release commands can be issued.

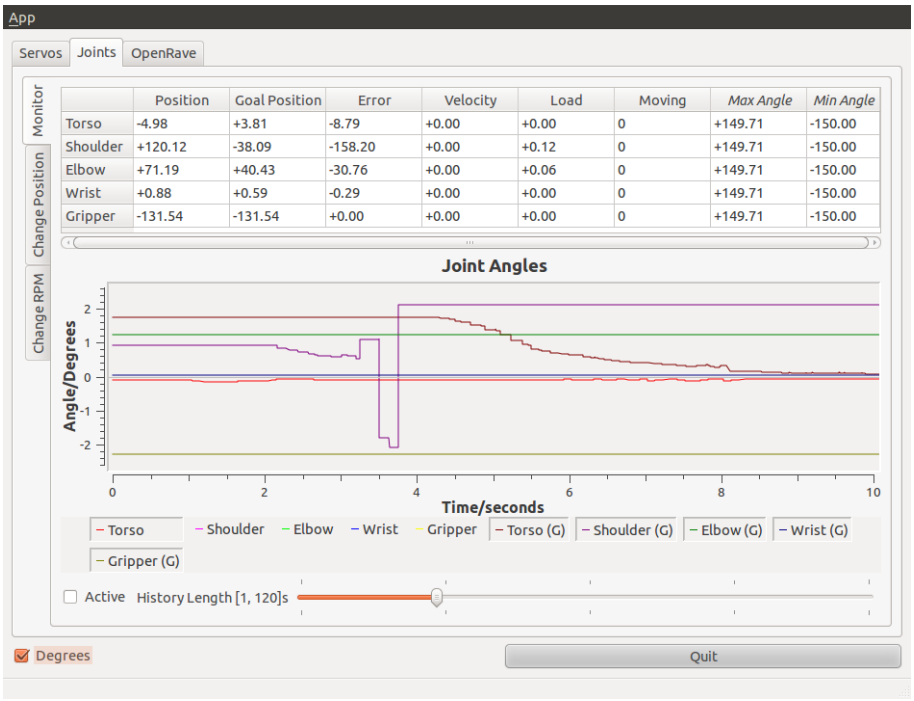


Figure 3.27: Individual joint details are updated at 10hz. Goal and current joint values can be plotted at different scales.

During the more comprehensive and longer "move chess piece from here to there and then return to an initial configuration" queries, progress bars and text boxes allow the user to observe the status and progress of planning and/or debug what is going wrong.

An example work flow might be the following: Using the translation visualization, the user can select objects and goal destinations. Then, using the joint state preview option, he/she can use the knobs to visually determine a joint configuration he would like the arm to always move to between moving chess pieces. Then the user can select the parameters and optimizations he/she wishes to use, select auto-execute and start planning. The MCN then does all the required tasks in conjunction with the ORN to move the selected piece to the given destination in simulation, and then execute the obtained trajectories on the real arm, while syncing the simulation to the real world, allowing the user to visualize discrepancies between the two. These sorts of work flows were essential to debugging the arm and fine tuning measurements, parameters and settings such as joint limits and offsets. A screen shot of every tab is shown in Figure 3.30, Figure 3.31 and Figure 3.32.

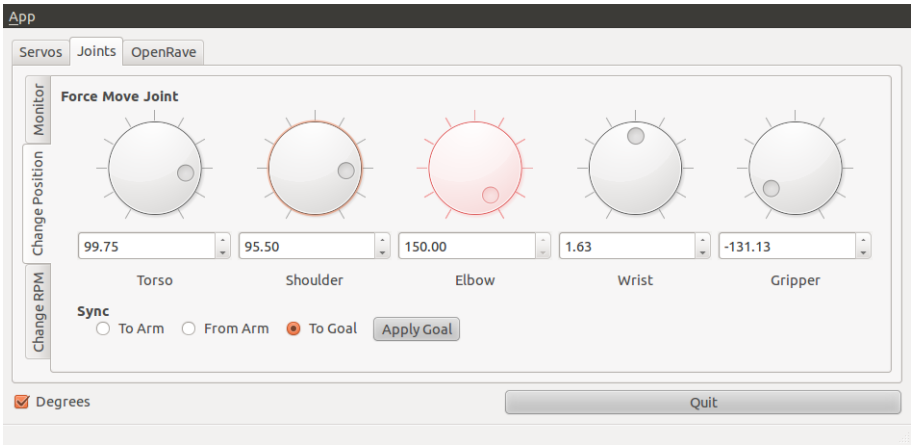


Figure 3.28: Individual joints can be graphically monitored and/or controlled. Knobs are highlighted when joint limits are surpassed

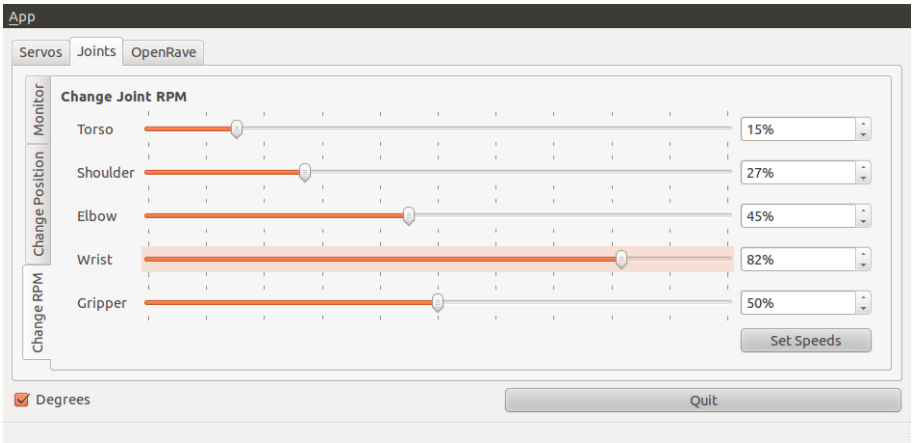


Figure 3.29: The SAC does not yet implement servo/joint speed changes, so this is taken from the diagnostic tool mentioned in Section 3.4.8 and not implemented here.

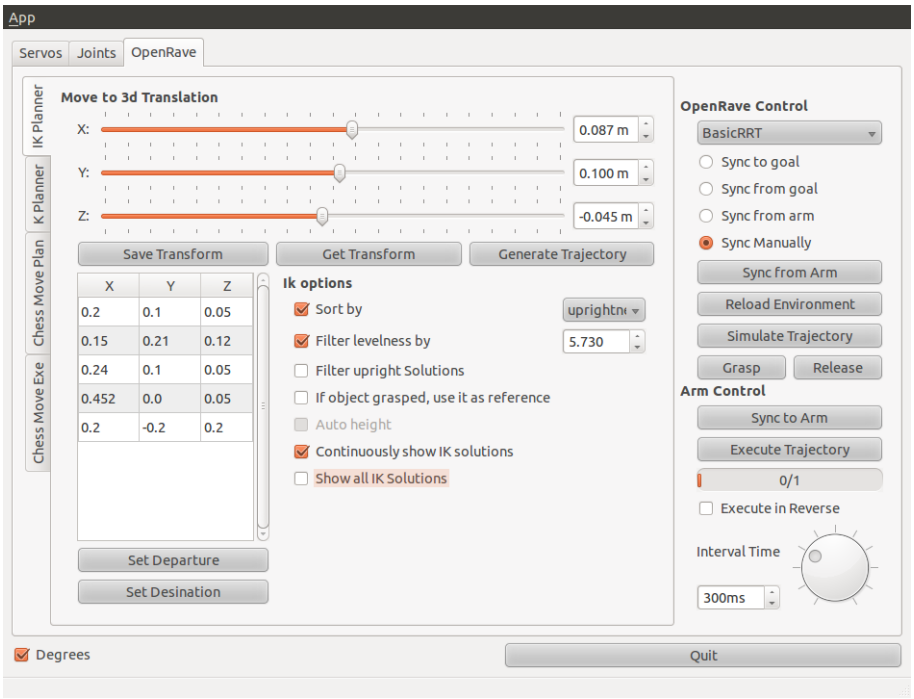


Figure 3.30: The user can chose transforms that are graphically displayed (not shown here) and invoke plan generation with different IK options. See Section 3.4.10

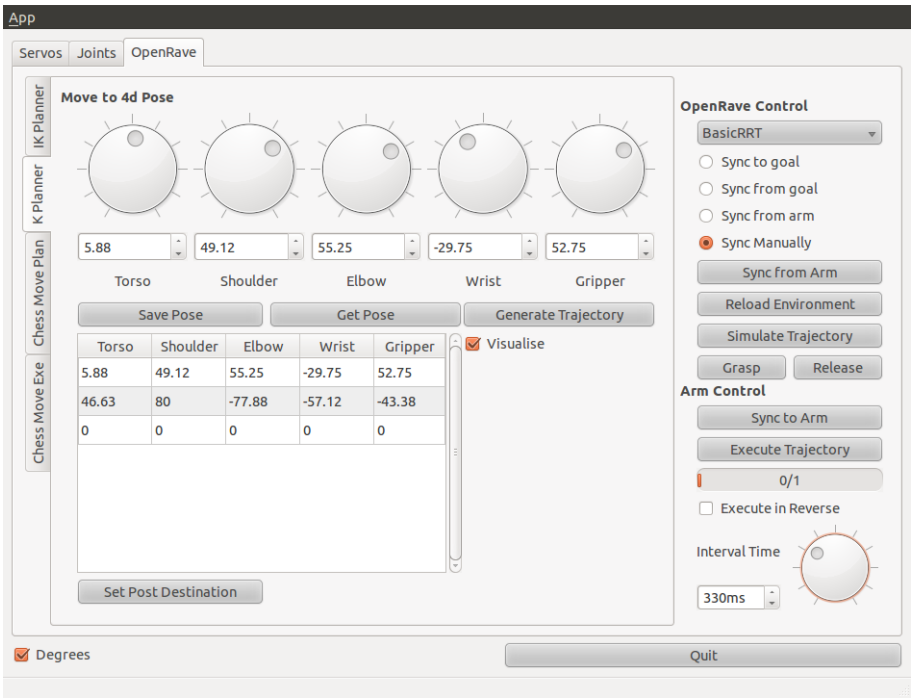


Figure 3.31: Joint configurations can be previewed and a plan can be generated to reach the specified preview. See Section 3.4.10

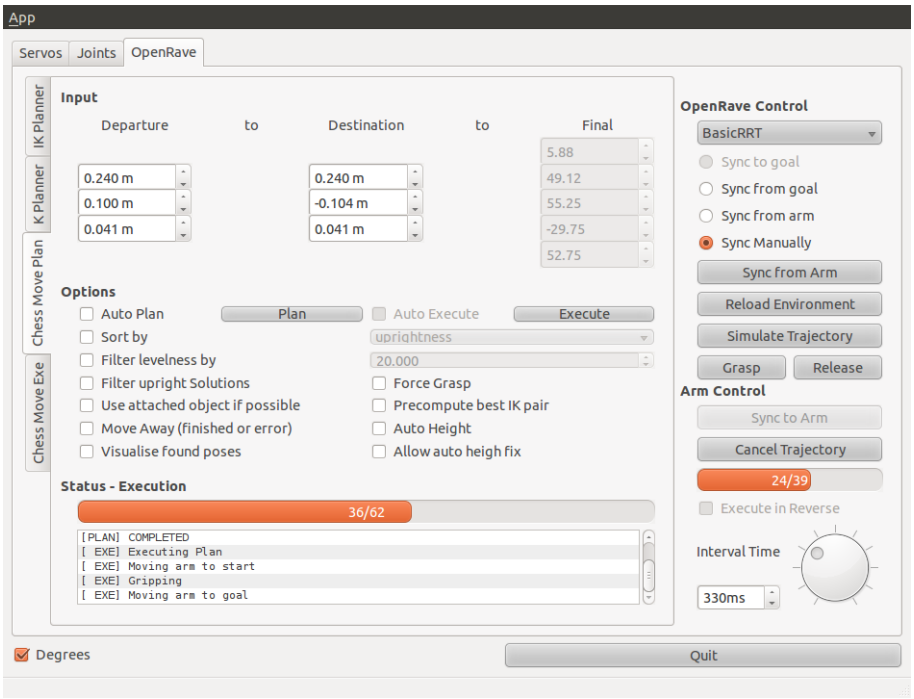


Figure 3.32: Entire move to target, grasp, move to goal, release, move to configuration type actions can be manually invoked here, or set to automatic. See Section 3.4.10

Chapter 4

Experiments and Results

In this chapter, we describe some of the experiments that we ran in a bid to evaluate the methodologies that we have proposed in Chapter 3 in Planning, Perception and Control. We also describe the results found and give illustration of the same. We discuss the implications of the said results on our methodology and highlight possible improvements when concluding in Chapter 5.

4.1 Perception

In order to put to test our methodology given in Section 3.3, we ran a series of experiments mainly categorized into the main methodology steps described in Section 3.3.1, Section 3.3.2 and Section 3.3.4.

4.1.1 Chessboard Detection

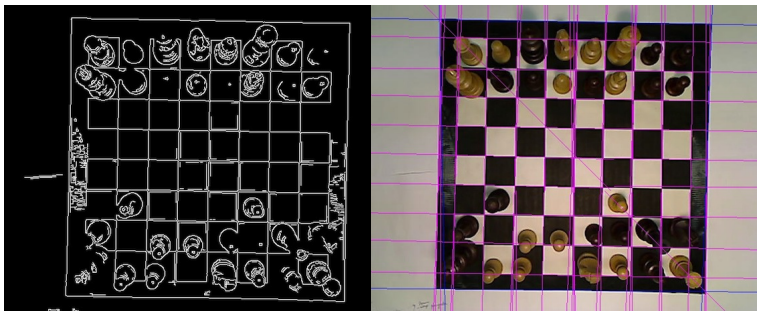
We show a sequence of processing from edge detection, line detection to the computation of bounding lines and their intersections in 4.1a and 4.1b.

4.1.2 Projecting the View

We also show a projected view of the chessboard from 4.1a and 4.1b on a 2D plane in Figure 4.2. It is from this projected image that we now perform change detection in order to determine user moves.

4.1.3 Move Detection

The move detection operation largely depends on the nature of the difference image[cite farahat] produced. Our current implementation requires that the difference image show two blobs indicating a move from one cell to another. This does not include certain chess



(a) Edge Detection **(b)** Line Detection

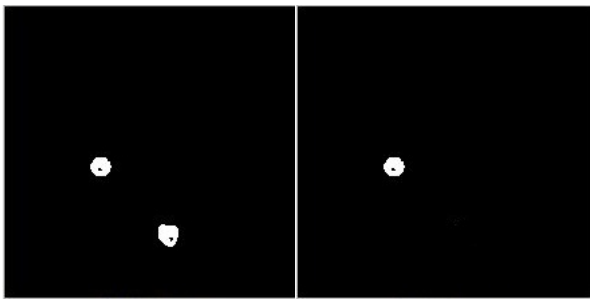
Figure 4.1: Edge and Line Detection

Figure 4.2: Chessboard view projected onto a 2D plane

moves in which chess pieces are swapped and taken off the board. However, extensions to our approach can easily made to accommodate such moves. One such a difference image is shown in 4.3a. Another equally important operation in the move detection is the flood filling of the blobs after detecting them by running distance transform algorithms. We show one such flood fill operation result in 4.3b. With these steps conquered the mapping of the coordinates of the blobs to 'chessboard dimensions' is a simple operation and the same for determining the moves using the game state matrix that we keep throughout the game.

4.1.4 Discussions

The implementation of the move detection is not complete in that not all chess moves can be detected, we however emphasize that the framework we have set up allows easy extensions to be able to detect such moves. Our implementation also experiences robustness issues with the chessboard detection and move detection. We attribute this to the rather raw implementation of the blob finding mechanism. We believe that use of more suitable blob finding APIs with our framework can improve this performance. The chessboard detection can be further improved by further tuning of the edge detection parameters as



(a) Difference Image **(b)** Flooded Image

Figure 4.3: Difference image and image after one round of floodfill

well as line detection parameters. We also envision experiments with basic tracking to investigate how this could affect the performance.

4.2 Manipulation

4.2.1 Experiments and Conclusions

Implementations and Optimisations

Experiments can be roughly divided into two types: those in the simulation world and those in the real world. For each point listed below, we highlight each type's results where applicable.

Levelness Figure 3.21 shows applying levelness in action. While levelness did not improve simulated performance, it was vital for real world application. Without levelness being enabled, the arm would fail to pick an object up most of the time. In addition, levelness greatly improved the synchronisation of the real world and the arm, as explained in Section 3.4.10. Sorting solutions by levelness means one needs to generate and then sort all Ik solutions, which usual means an increase of around 0.6-0.8 seconds to pick an optimal IK solution.

Uprightness Figure 3.24 shows the advantages of the using uprightness as a heuristic to place pieces well. As explains, it prevents objects from falling over when being placed. The benefits are two fold: (1) as the simulation does not support gravity, it was very important to try to place objects in such a way that the resulting transformation is as close to possible of the one the real world object would have after release, and (2) placing an object upright (obviously) improves the chances of it remaining upright - a required condition to place chess pieces. Picking the most upright IK solution takes 0.6-0.8 seconds longer then picking the first IK solution.

Pre-computing best IK pairs Figure 3.25 shows the difference between planning ahead and not. If one does not plan ahead, the likelihood of not being able to place an object upright increases. This was as important in the simulation as it was in the real world and turned out to greatly improve the overall chance of successful pick up and place operations, albeit at a slight cost in execution time. The time it takes to computing the best pair of IK solutions depends heavily on the number of solutions generated. For best results we filter non-level solutions out early (as explained in Section 3.4.5 and visualised in Figure 3.11), resulting in an average of $4 \cdot 4 = 16$ combinations to evaluate, meaning $1+4 = 5$ sets of IK solution need to be generated resulting in an increase of 6-12 seconds over picking the first available IK solutions.

Autoheight enabled through planning in the attached object frame Figure 3.23 demonstrates planning in the attached object frame by showing the `autoheight()` function in actiong. This greatly improved the accuracy of releasing an object at a given location and helped the simulation keep the pieces low over successive pickup and drops with minimal user interaction and tweaking. Overall, this worked out very well in practise.

Planning Figure 3.12 demonstrates a generated plan. In our chess playing case, there are usually no large/complicated obstacles in the way so that planning works very fast. To generate a plan to move the EEF from one cell to another would take less than 0.5s, and a plan to move from one corner of the board to the opposite would take less then 4 seconds. When many small obstacles clutter the start / goal area, plan generation is slowed down due to collision checking (can take 1-3 times as long). Using cylinders instead of detailed chess models greatly improves performance by several magnitudes (plan generation took minutes). I do not recall the planner failing to find a solution if one existed. Often, the arm could not release objects within an uprightness threshold, as too few IK solutions existed. This limited the ability of the arm to successfully place chess pieces.

Hardware Control and Performance Hardware control worked out well. The SAC communicated with the arm well, resulting in fluid movement whens the trajectory interval was set between 30ms (fast) and 50ms (fluid). Servo readings were accurate and the Dynamixel network dropped very few packets. The power supply was sufficient. However, as mentioned in Section 3.4.8, the torso servo could not sustain high angles and would simply deactivate. This strongly affected the performance of the arm as plans that involve the arm stretching it would simply fail, resulting in an manipulator that could not recover without user intervention. Gravity also had a strong affect on the servo cause a large mis-match between the simulated arm and real arm, often causing collisions in the real world that were avoid in the simulated one. In addition, the gripper was very difficult to work with. It grasped objects with unpredictable results, often displacing/rotating them in the process, such that uprightness optimisations were flawed as the calculations were done with transforms that were not really the case.

Environment Section 3.4.7 shows tests of reachability as shown in Figure 3.11 and Table 3.2, where the general set-up of the environment and arm were justified. The environment construction was rock solid and did not vibrate and was simple enough to be accurately modelled. The arm could reach all the desired cells.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Perception We have demonstrated that the constraint of using a modified chessboard and/or pieces can be relaxed without compromising performance. We have also demonstrated that the constraint of having a fixed camera directly above the chessboard can be done away with while still enjoying reasonable real-time performance. We have shown all of these using very simple hardware and a limited amount of time. Our findings can therefore serve as a proof of concept for further investigation using more robust hardware.

Manipulation In short, the simulated world worked out well, but the real world tests did not. The optimisations described in this thesis greatly increase the usability of the arm in theory (i.e. in simulation) but the arm hardware is still too imprecise to allow the optimisations to work well. In addition, the too few DOFs meant that pieces could not be place in an upright fashion, meaning they always fell over. Due to these limitations, a successful game of chess could not be played.

5.2 Future Work

Perception In the future, we would like to investigate the effect of adding an object recognition pipeline to our framework to aid in determining the orientations of the chess pieces on the board. We would also like to investigate the effect of adding additional sensors to our perception pipeline. These could include using multiple cameras or just using different additional sensors like the Kinect. As mentioned in Section 4.1.4, we would also like to add more suited blob detection mechanisms to our pipeline.

Manipulation First, one would have to upgrade the servos to more precise and strong ones. Dynamixel AX-18F servos would be an easy swap as they have the same dimensions

and communication protocol, but whether they suffice would have to be tested. Further more, one would need to increase the DOF of the arm. An additional joint before the wrist rotation around the local Z axis would probably suffice to greatly increase the "upright" reachability of the arm, allowing it to place objects upright independent of where the object was initially grasped. Ultimately, the arm chosen arm is unsuited for the task. A much simpler 3DOF manipulator with only translational actuators would be much more suited, cost efficient, elegant or simpler to implement. However, deciding to solve a problem with unsuited means gives incentive to overcome some of the shortcomings with creative improvements. Tighter integration of the ORN and MCN could facilitate user interaction and cosmetic changes, while simultaneously reducing overhead over the ROS network.

Writing of the chapters was split as follows:

Chapter	Written by
Introduction	Oliver Dunkley
State of the Art	Billy Okal
Methodology: Core Control	Oliver Dunkley
Methodology: Perception	Billy Okal
Methodology: Planning	Oliver Dunkley
Experiments: Perception	Billy Okal
Experiments: Planning	Oliver Dunkley
Conclusion	Billy Okal and Oliver Dunkley

Bibliography

[1] ICRA 2010 and AAAI 2010. Small-scale manipulation challenge: Table top chess. `http://aaai-robotics.ning.com/forum/topics/icra2010-and-aaai2010`.

[2] Rosen Diankov. *Automated Construction of Robotic Manipulation Programs*. PhD thesis, Carnegie Mellon University, Robotics Institute, August 2010.

[3] F.C.A. Groen, G.A. der Boer, A. van Inge, and R. Stam. A chess playing robot: lab course in robot sensor integration. In *Instrumentation and Measurement Technology Conference, 1992. IMTC '92., 9th IEEE*, pages 261 –264, May 1992.

[4] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521623049, 2000.

[5] Intel. Open source computer vision library. `http://opencv.willowgarage.com`.

[6] J.J. Kuffner Jr and S.M. LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE, 2000.

[7] Tekkotsu Labs. Chiara robot. `http://chiara-robot.org/`.

[8] J. C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[9] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[10] S.M. LaValle. Rapidly-exploring random trees: A new tool for path planning. *In*, (98-11), 1998.

[11] Monty Newborn and Monroe Newborn. *Kasparov Vs. Deep Blue: Computer Chess Comes of Age*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[12] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[13] Nathan Ratliff, Matthew Zucker, J. Andrew (Drew) Bagnell, and Siddhartha Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In *IEEE International Conference on Robotics and Automation (ICRA)*, May 2009.

[14] IDL Reference. Idl reference guide: Hough. `http://idlastro.gsfc.nasa.gov/idl_html_help/HOUGH.html`.

[15] CrustCrawler Robotics. Ax-12+ smart robotic arm. `http://www.crustcrawler.com/`.

[16] PAL Robotics. Reem-a. `http://www.pal-robotics.com/robots/reem-a`.

[17] Simon Schaffer. Enlightened automata. In *The Sciences in Enlightened Europe. (Eds. William Clark, Jan Golinski, and Simon Schaffer)*, pages 126– 165, 1999.

[18] Intel Labs Seattle. Gambit : A chess-playing robot. `http://ils.intel-research.net/gambit`.

[19] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image Processing, Analysis, and Machine Vision*. Thomson-Engineering, 2007.

[20] David Urting and Yolande Berbers. Marineblue: A low-cost chess robot. In *IASTED International Conference on Robotics and Applications (Hamza, M.H., ed.)*, pages 76 –81, 2003.

Proclamation

We Hereby confirm that we wrote this thesis independently and that we have not made use of any other resources or means than those indicated.

Bremen June 2, 2011