Bachelor thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Playing chess with KUKA robot using linguistic instructions

Marek Jalůvka

Supervisor: Mgr. Karla Štěpánová, Ph.D., Mgr. Gabriela Šejnová
May 2019

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Jalůvka Marek**	Personal ID number:	**466227**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Cybernetics and Robotics**		

II. Bachelor's thesis details

Bachelor's thesis title in English:

Playing Chess with KUKA Robot Using Linguistic Instructions

Bachelor's thesis title in Czech:

Hra šachů s KUKA robotem pomocí jazykových instrukcí

Guidelines:

1. Familiarization with robotic setup, ROS, packages for natural language processing (e.g. speech recognition) and object detection from RGBD camera.
2. Preparation of the experimental setup - chessboard, objects for chess playing, cameras, microphone etc.
3. Object detection, grasping and movement of figures to the position on the chessboard determined by linguistic instructions.
4. Detection of the figures which should be removed by the given move from the chessboard.
5. Identification of illegal and disadvantageous moves (e.g. check/checkmate). Robot should communicate back to the player.
6. Optional incorporation of automatic chess playing algorithm.

Bibliography / sources:

[1] Matuszek, Cynthia, et al. - "Gambit: An autonomous chess-playing robotic system." - Robotics and Automation (ICRA), 2011 IEEE International Conference on. IEEE, 2011.
[2] Dantam, Neil, and Mike Stilman. "The motion grammar: Analysis of a linguistic method for robot control." IEEE Transactions on Robotics 29.3 (2013): 704-718.
[3] Gonçalves, José, José Lima, and Paulo Leitao. - "Chess robot system: A multi-disciplinary experience in automation." - 9th Spanish Portuguese Congress On Electrical Engineering. 2005.
[4] Chen, Andrew Tzer-Yeu, I. Kevin, and Kai Wang. / "Computer vision based chess playing capabilities for the Baxter humanoid robot." / Control, Automation and Robotics (ICCAR), 2016 2nd International Conference on. IEEE, 2016.
[5] Cour, Timothée, Rémy Lauranson, and Matthieu Vachette. "Autonomous chess-playing robot." Ecole Polytechnique, July (2002).

Name and workplace of bachelor's thesis supervisor:

Mgr. Karla Štěpánová, Ph.D., Robotic Perception, CIIRC

Name and workplace of second bachelor's thesis supervisor or consultant:

Mgr. Gabriela Šejnová, Robotic Perception, CIIRC

Date of bachelor's thesis assignment: **24.01.2019** Deadline for bachelor thesis submission: **24.05.2019**

Assignment valid until: **20.09.2020**

Mgr. Karla Štěpánová, Ph.D.	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Ing. Pavel Ripka, CSc.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

I would like to thank Mgr. Karla Štěpánová, Ph.D. for great guidance and valuable advice. I am also grateful to Mgr. Gabriela Šejnová for revision of this thesis. Many thanks to Mgr. Michael Tesař for the help with preparation of the experimental setup. I would like to thank Ing. Vladimír Smutný, Ph.D and Ing. Libor Wagner for the design, manufacturing and integration of the electromagnetic gripper with the robot. At last but not least, I would like to thank my family for unwavering support not only during the writing of this thesis.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, May 23, 2019

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 23. května 2019

Abstract

This thesis describes playing chess with KUKA robot using linguistic instructions. First, the individual technologies used for this project are described including the robotic arm, voice recognition software and object detection framework. Second, the whole setup assembled for this purpose and the communication between its components is explored. Third, an overview of the integrated chess logic is provided. Finally, the whole architecture is tested and evaluated.

Keywords: chess-playing, KUKA, robot, voice control, ArUco, object detection, ROS

Supervisor: Mgr. Karla Štěpánová, Ph.D., Mgr. Gabriela Šejnová

Abstrakt

Tato práce popisuje hraní šachů s KUKA robotem pomocí jazykových instrukcí. Nejprve jsou popsány jednotlivé technologie použité pro tento projekt, včetně robotické ruky, softwaru pro rozpoznávání hlasu a systému pro detekci objektů. Dále je zkoumán systém vzniklý pro uskutečnění tohoto projektu a komunikační rozhraní jeho komponent. Poté je poskytnut přehled integrované šachové logiky. Nakonec je celá architektura otestována a ohodnocena.

Klíčová slova: hra šachů, KUKA, robot, hlasové ovládání, ArUco, detekce objektů, ROS

Překlad názvu: Hra šachů s KUKA robotem pomocí jazykových instrukcí

Contents

Figures

Tables

Chapter 1
Introduction

As the modern technology keeps advancing and robots' range of capabilities expands, the idea of a robotic companion cooperating with humans outside of industrial environment seems more and more plausible. In contrast with contemporary robots, the robotic butlers, teachers or companions for the elderly of the future need to interact naturally with people - i.e. gesture, posture or language control. Even untrained personnel have to be able to operate the robot. The quality of this interaction is a major factor influencing how well the robotic system will be accepted, because humans are very sensitive to communication delays, low responsiveness and unpredictable behaviour. The robot should therefore not only listen to commands, but respond seamlessly with its understanding of the task at hand.

One of the ways robots could find a place in a common household is by playing games with the humans. For those applications, robots have to react to human behaviour, perceive its surroundings, mainly the objects involved within the game and finally understand the game itself. This requires integration of various technologies like camera image processing, speech recognition and robot hardware capable of fine motoric movements.

Such consumer games currently available use mostly hard-coded object locations and show only a minimal amount of flexibility. Voice control is not nearly as common as it should be by now.

For this thesis, our game of choice is chess. It has clear rules, observable game state and the moves can be executed by a robot. These features make it a well-suited target application for this thesis.

Goal of this thesis

This thesis aims to create and describe a framework for communicating with KUKA robot for the specific purpose of playing chess. Design of the game and all the technologies used in the process are also in the scope of this thesis. Despite a big portion of the code being task specific, some functions and concepts can be reused for other similar applications. That is mainly pick and place tasks with object detection. The same approach can be also applied to a voice control of an industrial robot, i.e. teleoperation.

The main goal here is to integrate visual input with natural language processing and use both of them to control a robotic arm. The visual information helps to create abstract representation of the real world setup - the position of chess pieces in this case - while high level linguistic commands tell the robot which chess move to perform.

Aside from this, I aspire to create an interactive game to play with the robot such that it can show off robot's accurate motion capability, a way to perform object detection and speech recognition all while entertaining the human player as well.

Contribution

The main contributions of this thesis are:

- assembling an experimental setup for chess-playing including the design and manufacturing of chess pieces,
- creation of a framework for robot-human interaction - integration of visual and linguistic input with robot control,
- creation of basic chess logic - move validity check, resolution of ambiguities, etc,
- voice or textual communication interface,
- evaluation of the whole system.

Chapter 2
Related work

There have been many successful attempts to realize human - robot chess playing. For example Raspberry Turk by J. Meyer [Mey17] is an open source chess-playing robot. According to its website, it was named after the Mechanical Turk, which was a machine constructed in the late 18th century and had become famous for being able to play chess against a human opponent and beat them most of the time. [Fou19] The main principle of this machine turned out to be an advanced neural network: a skilled chess player was hidden inside. On the other hand, Raspberry Turk is run on Raspberry Pi, written mostly in Python. The robotic arm has two rotational and one linear joints, while the end-effector is an electromagnet. Special 3D-printed chess pieces are used for this application. Camera above the chessboard helps to determine the position of the chess pieces, or more accurately the change from the previous state. Assuming only one piece was moved, only a "simple" algorithm needs to be implemented to determine, whether a particular chess field has a piece on it and what color it is. The main hiccup here is dealing with the situation of a promotion of a pawn to another chess piece and recognizing what piece it is (queen is not always the best option). For this task the author uses convolutional neural networks.

Another interesting approach to this task is project by Convens, et al [BCW17], where the authors created a chess-playing robot using two linear actuators underneath the chessboard providing two degrees of freedom. The pieces were moved by an electromagnet attracted to neodymium permanent magnets at the bottom of the pieces. An optical character recognition (OCR) algorithm, Tesseract more specifically, was run on the camera input data to detect position of individual pieces inside chessboard fields. An open source chess playing software Stockfish provided artificial intelligence behind the logic of the robot moves.

Al-Saedi and Mohammed [ASHM15] utilized Lab-Volt 5150 robotic arm for playing chess. In the paper, the manipulator is described in great detail including forward and inverse kinematics. Chess pieces are detected here using a custom built smart chessboard consisting of 2D array of reed switches that are normally open, but close upon applying magnetic field provided by a ring magnet installed at the bottom of all pieces. With this setup, it is possible to detect only whether there is a piece or not on a particular field.

Similarly to [Mey17], only changes from the last configuration are tracked throughout the game.

B. Yeh, A. Trakowski, D. P. Martin and J. Flohr [BYF13] built a voice controlled chess playing robot - a 3 DOF cartesian construction with servo motors controlling the axes and built-in encoders in two of them. The z position was determined by a sensor measuring distance from the chessboard. The gripper was a mechanical claw. The pieces were moved by the robot exclusively, so no object detection was needed given correct initial position. The "`chess_at_nite`" engine performed all game logic including move validation resolution, check/mate detection and even provided an automatic chess playing algorithm. As for the voice control, EasyVR module was used. The commands were provided in the "source field" - "target field" form to avoid ambiguities. The VR system also needed to be trained on every user for each utilized word. For better results, Navy phonetic alphabet replaced the classical one - e.g. "Bravo 4 to Alpha 3".

N. U. Alka, A. A. Salihu, Y. S. Haruna and I. A. Dalyop [NUAD17] created voice controlled vehicle with a robotic arm for pick and place applications. For reception and processing of the language commands, AMR Voice and Google Voice Search are used. The final commands are send over Bluetooth to ATMEGA328P microcontroller, which is a part of the vehicle and drives the motors.

In paper by G. Bohouta and V. Këpuska [BK17], different software for speech recognition is compared, Microsoft API, Google API and Sphinx-4, more specifically. Google API is in the end considered to be the best option – it had the lowest WER (word error rate).

Chapter 3
Materials and Methods

Experimental setup

Main components of this project are LBR KUKA iiwa 7 robotic arm, Intel RealSense camera, a PC running Linux with microphone and a custom built set for playing chess. The pieces are picked up by an electromagnet mounted on the robot end-effector. The pivotal software used to integrate all of the above into one project is Robot Operating System (ROS). ROS `real_sense2_camera`[Dor19] package provides basic RealSense camera data extraction, `tuw_marker_detection`[Bad18a] package detects ArUco markers from the camera data, `language_ctrl`[Še18] is used for speech recognition and finally `capek_testbed` [VP18] controls the robotic arm. Outside ROS packages, OpenCV handles camera calibration and Eigen library performs matrix operations for the purpose of spatial transformations. Lastly, RViz is a convenient ROS-integrated simulator with many visualization options e.g. markers and marker arrays included in this project. All of these components will be described in a greater detail within this chapter.

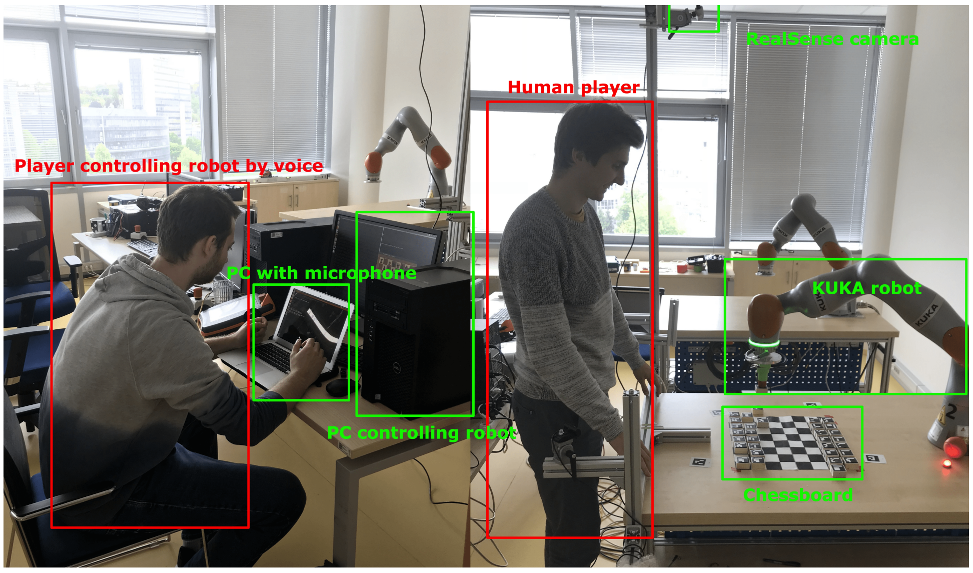


Figure 3.1: Experimental setup

Chess pieces

The final iteration of chess pieces used for this game are wooden blocks with dimensions 41x41x20 mm. A steel plate 30x30x0.75 mm was glued to one of the bases. On top of it, an ArUco marker printed on self-adhesive paper was sticked. On a side of each piece, its respective type was sticked as well. A photo of a black pawn is in Figure 3.2, while the whole collection on the chessboard is depicted in Figure 3.3.

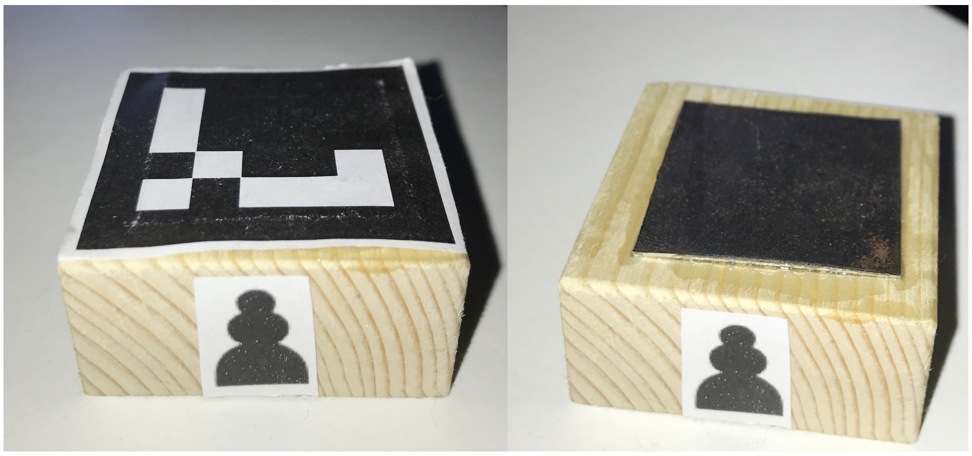


Figure 3.2: A black pawn with and without ArUco marker on top.

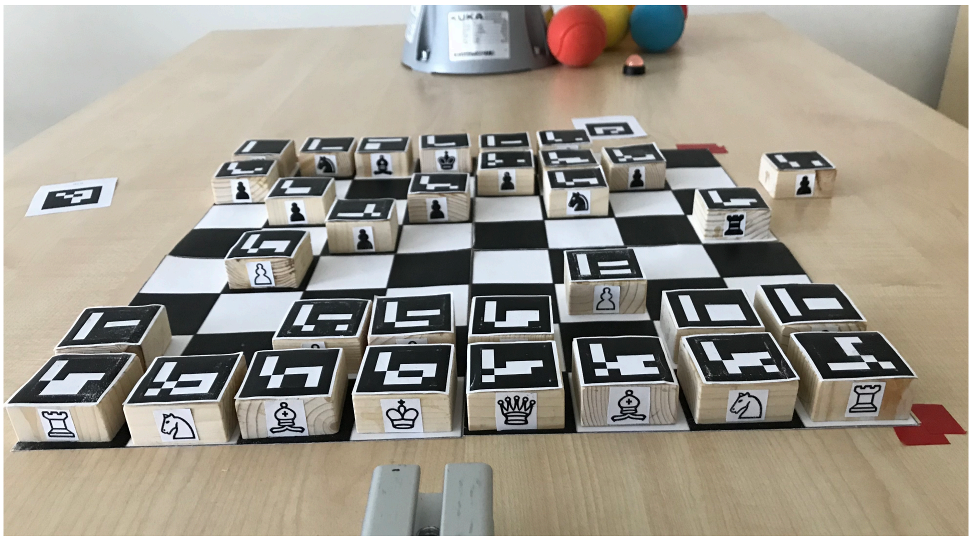


Figure 3.3: The whole chess set.

ROS

ROS is an open source framework for developing robotic applications as it contains many useful libraries and tools for integrating various technologies, visualization and debugging. There is an abstraction layer added for the purpose of communication between different parts of the running program on one machine or multiple machines via a network. Thanks to this, ROS behaves like a distributed system even when run on a single PC.

The ROS environment comprises of ROS packages containing ROS nodes - executable programs written mainly in C++ or Python (but Java, MATLAB or Lisp are supported as well). A central node called ROS Master is needed running at all times as it is used to initiate communication between any other nodes, keeps track of node addresses and provides parameter server. The initialization process of a topic subscriber and publisher is depicted in Figure 3.4. Once initialized, nodes communicate directly, i.e. point-to-point model is used.

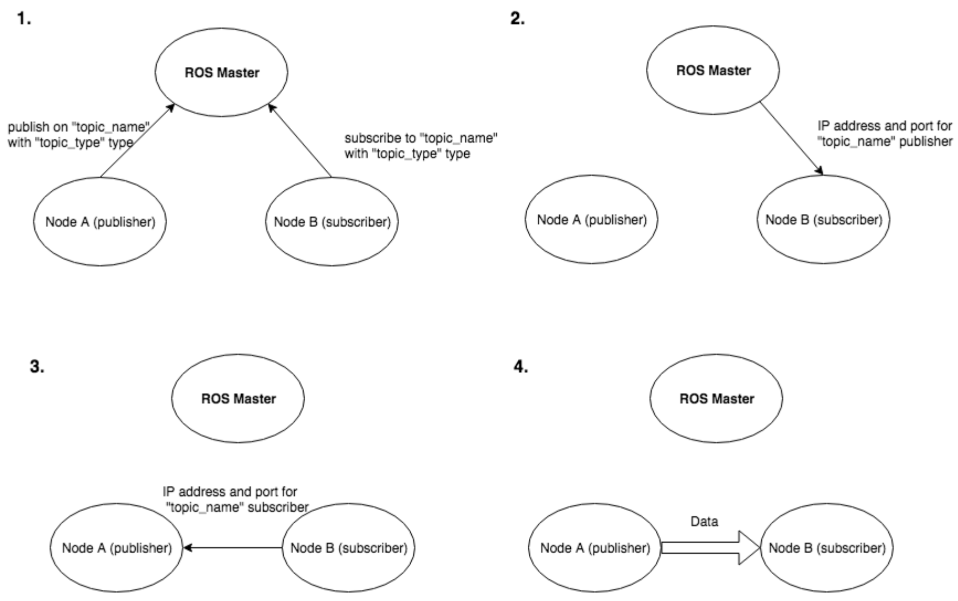


Figure 3.4: Initialization of communication via a ROS topic. **1.** Registration with the Master, **2.** Master sends contact information for publisher to the subscriber, **3.** subscriber contacts the publisher directly with its contact information, **4.** publisher sends data to subscriber, drawn using https://www.draw.io/.

ROS topics are asynchronous, uniquely named, communication channels designed for connecting multiple nodes. Each topic supports only a particular type of ROS message. This type can be a concatenation of simpler types. ROS `std_msgs` package contains some basic message types including Bool, Char, Int8, Float32, etc. Some of the more advanced types needed for this project include `geometry_msgs/Pose`, `visualization_msgs/Marker` and `marker_msgs/MarkerDetection`. Their definitions are provided below.

Listing 3.1: `geometry_msgs/Pose` [Foo18]

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

Listing 3.2: `visualization_msgs/Marker` [Fau18]

```
uint8 ARROW=0
uint8 CUBE=1
uint8 SPHERE=2
uint8 CYLINDER=3
uint8 LINE_STRIP=4
uint8 LINE_LIST=5
uint8 CUBE_LIST=6
uint8 SPHERE_LIST=7
uint8 POINTS=8
uint8 TEXT_VIEW_FACING=9
uint8 MESH_RESOURCE=10
uint8 TRIANGLE_LIST=11
uint8 ADD=0
uint8 MODIFY=0
uint8 DELETE=2
uint8 DELETEALL=3
std_msgs/Header header
string ns
int32 id
int32 type
int32 action
geometry_msgs/Pose pose
geometry_msgs/Vector3 scale
std_msgs/ColorRGBA color
duration lifetime
bool frame_locked
geometry_msgs/Point[] points
std_msgs/ColorRGBA[] colors
string text
string mesh_resource
bool mesh_use_embedded_materials
```

Listing 3.3: `marker_msgs/MarkerDetection` [Bad18b]

```
std_msgs/Header header
float32 distance_min
float32 distance_max
float32 distance_max_id
geometry_msgs/Quaternion view_direction
float32 fov_horizontal
float32 fov_vertical
string type
```

```
marker_msgs/Marker[] markers
```

Another way to communicate within the ROS infrastructure is using ROS services. In contrast with topics, they perform synchronous remote procedure calls and are therefore based on the request - response (or client - server) model. In this project, a service call turns the robot mounted electromagnet on and off. [Šk17] [AB16]

Eigen library

Eigen library is an open source cross-platform high-level C++ library, that contains functions useful for linear algebra like matrix and vector operations and geometrical transformations. Numerical solvers are included as well. The basis of its functionality are C++ expression templates. C++ templates are a metaprogramming technique for creating generic data types and subsequently functions, whose input arguments and return value can be different data types according to the particular function call. Generic types are specified by the compiler. The following code creates a generic function for adding numbers (the compiler substitutes int types in this case):

Listing 3.4: Templates usage example

```
using namespace std;

template <class T>

T add(T a, T b)
{
    return a+b
}

int main()
{
    int x = 1, y = 2, z;
    z = add(x, y);
}
```

Furthermore, expression templates are used to create structures representing computations built at compile time. The goal here is to make the evaluation of an expression as efficient as possible, for example loop fusion is achieved by this method. When presented with a task such as summing vectors:

$$\boldsymbol{z} = \boldsymbol{u} + \boldsymbol{v} + \boldsymbol{w} + \boldsymbol{x}, \tag{3.1}$$

instead of summing $\boldsymbol{tmp1} = \boldsymbol{u} + \boldsymbol{v}$ first, then $\boldsymbol{tmp2} = \boldsymbol{tmp1} + \boldsymbol{w}$ and finally $\boldsymbol{z} = \boldsymbol{tmp2} + \boldsymbol{x}$, this approach allows for restructuring the source code by the compiler such that this summation is performed in one loop instead of three with no need to use extra memory (**tmp1** and **tmp2** in this case).

The Eigen library supports every type of common matrix and vector operation including block operations, broadcasting (replicating a vector in one direction to represent a matrix), reshaping and slicing. Various matrix decompositions are implemented here as well, e.g. LU, QR, SVD, eigendecomposition and much more. With these, systems of linear equations can be solved or approximated by the least square method when the solution doesn't exist. Finally, spatial and planar transformations can be computed with Eigen - 2D and 3D rotations, translations and scaling. Rotation matrix, quaternions, angle-axis and Euler angles are supported rotation representations.

The Eigen library was used in the C++ part of this project primarily for convenient creation (the << operator) and multiplication of rotation and transformation matricies, as well as functions for conversion between rotation matrix and quaternion rotation representation. [GJ+10] [Vel94]

Camera calibration using OpenCV

For camera calibration, the OpenCV library was used. OpenCV is an open source library with focus on computer vision and machine learning. According to its website, OpenCV has more than 2500 optimized algorithms that can be used to "detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality, etc" (`https://opencv.org/about/`, April 2019) [Its14] [Its15].

The extrinsic parameters (position and orientation in world coordinate frame) of Intel Realsense camera capturing the scene for this project have been calibrated using two different methods that both operate on OpenCV basis - robotic arm ArUco marker calibration and chessboard calibration.

During the first method, an ArUco marker was placed at the robot end-effector and was detected by the camera in multiple poses of the robotic arm. By this procedure, corresponding pairs of points were obtained - robot end-effector poses on the one hand and ArUco marker poses in camera coordinate system on the other hand.

The second method involved only a chessboard with known dimensions and position in the world coordinate system. `chessboard_camera_calibration` node in `robot_chess_player` package performs this calibration. This node listens to `"camera/color/image_raw"` topic (Realsense camera publishes image data here), then uses the `cv_bridge` ROS package to convert images from ROS to OpenCV format. On each frame (after grayscaling) cv2.findChessboardCorners function is called. This function returns locations of chessboard corners (who would have guessed) on that frame. For more precise corner positions, cv2.cornerSubPix is called on the obtained corners.

Having enough 3D - 2D point correspondences (49 in our case), camera extrinsic parameters can be calculated by cv2.solvePnP function. OpenCV implements this function (and many more for that matter) assuming pinhole camera model. This model transforms points from the world coordinates to camera image plane like so:

$$\begin{aligned}\lambda \boldsymbol{x} = \lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} &= \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}, \\ &= \boldsymbol{K}[\boldsymbol{R} \quad \boldsymbol{t}]\boldsymbol{X_w}\end{aligned} \tag{3.2}$$

where $\boldsymbol{K}$ is a matrix of intrinsic parameters, f_x and f_y are focal lengths expressed in pixel units and (c_x, c_y) is a principal point usually at the center of the image. This matrix can be further decomposed to

$$\boldsymbol{K} = \begin{bmatrix} s_x & 0 & c_x \\ 0 & s_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.3}$$

where f is the lense focal length (basically the distance from the lense at which the image plane should be placed) and s_x and s_y are sizes of the individual image elements in pixels/milimeter in that particular direction. This matrix can be also determined by calibration, but in our case was provided by the manufacturer of the camera.

The goal of this calibration was to obtain the matrix of extrinsic parameters $[\boldsymbol{R} \quad \boldsymbol{t}]$, so by looking at 3.2, this can be achieved with at least 5 correspondences of points in the world coordinates $\boldsymbol{X_w}$ with their pixel coordinates $\boldsymbol{x}$, as there are 13 variables (all the $[\boldsymbol{R} \quad \boldsymbol{t}]$ matrix elements and λ) and 3 equations are provided with each correspondence. The Figure 3.5 taken from the OpenCV Camera Calibration and 3D Reconstruction documentation illustrates this model's basic concept. Usage of the first camera calibration method resulted in rather inaccurate outcome - the error between the position of the detected and the real chess pieces was up to two chess fields. This was the initial motivation for the second method. Here, pieces on the left down corner of the chessboard (from the perspective of the human player) were placed correctly, but the further right and up a piece was placed, the worse the error got. To correct this last discrepancy, a chess piece (with an ArUco marker) was placed to each chess field with known world coordinates and the position it actually detected and calculated was saved to a file. This way, a dataset of corresponding points was collected. The next task was to create a function, that takes in the mostly incorrect calculated positions of the chess pieces and returns correct ones. Because of the nature of the problem, the functions were expected to be affine:

$$\begin{aligned} x_{corr} &= f(x_{det}, y_{det}) = a x_{det} + b y_{det} + c \\ y_{corr} &= g(x_{det}, y_{det}) = d x_{det} + e y_{det} + f, \end{aligned} \tag{3.4}$$

where (x_{det}, y_{det}) are the detected positions and (x_{corr}, y_{corr}) are the corrected positions. The coefficients a, b, c, d, e and f had to be found. Most likely, for

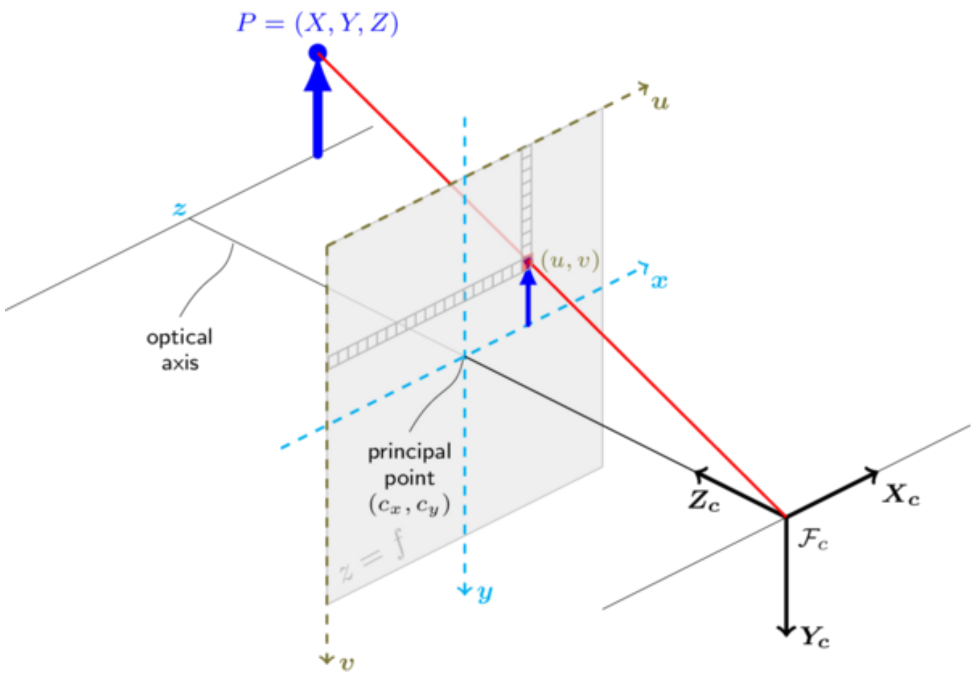


Figure 3.5: Pinhole camera model, from [ope19].

all measured data points there will be no exact solution, but rather difference between the function outcome and the correct points must be minimized, therefore this is a linear regression problem. Formally, we are looking for:

$$\min_{\boldsymbol{x}=(a,b,c)} ||A\boldsymbol{x} - \boldsymbol{b}|| = \min_{\boldsymbol{x}=(a,b,c)} \left\| \begin{bmatrix} x_{det1} & y_{det1} & 1 \\ x_{det2} & y_{det2} & 1 \\ \cdots & \cdots & \cdots \\ x_{detn} & y_{detn} & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} - \begin{bmatrix} x_{corr1} \\ x_{corr2} \\ \cdots \\ x_{corrn} \end{bmatrix} \right\|, \tag{3.5}$$

where $(x_{deti}, y_{deti}), i = 1, \ldots, n$ are the measured data points and x_{corri} are the correct values we are trying to approximate. For this least square problem, a closed form solution exists:

$$\boldsymbol{x} = (A^T A)^{-1} A^T \boldsymbol{b}. \tag{3.6}$$

Similar way, the d, e and f coefficients were found.

After applying this correction, the accuracy of virtual chess piece placement improved to a satisfactory level.

ArUco markers and `tuw_marker_detection` package

ArUco markers are square-shaped markers consisting of a black border and black and white array of squares inside it representing a binary information. An ID is assigned to each marker according to its index in the particular marker dictionary (dictionary being a set of markers grouped together usually with

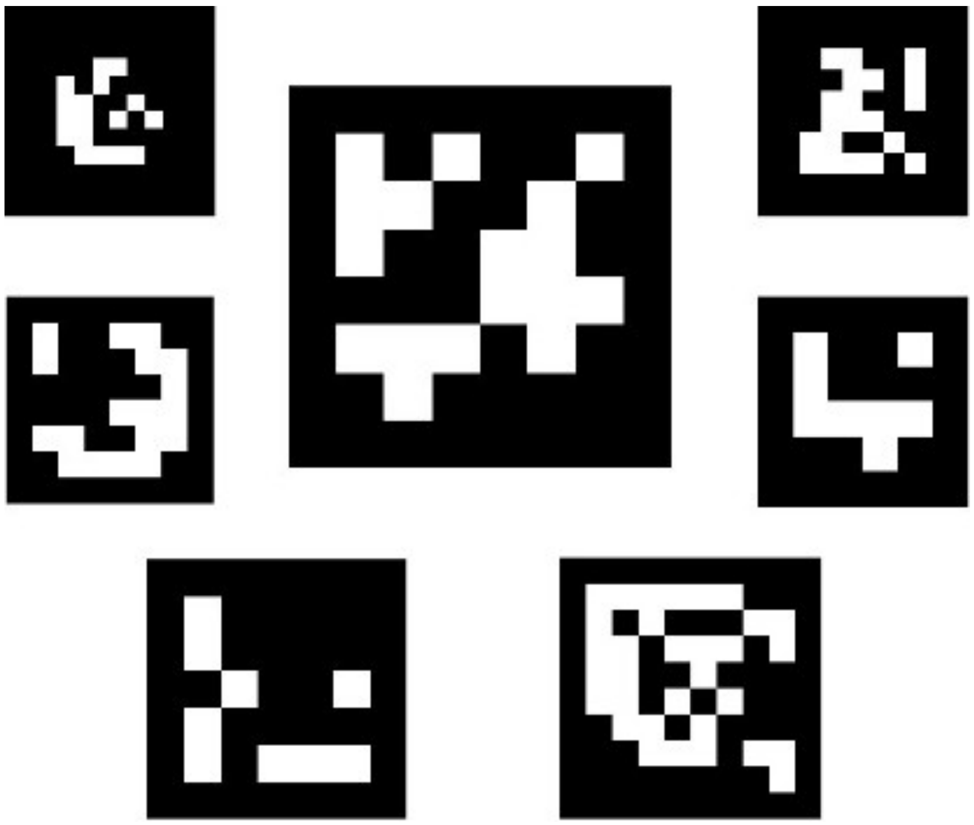


Figure 3.6: Examples of markers, from [mar15].

the same size). The `tuw_aruco` node inside `tuw_marker_detection` package is a ROS wrapper around the augmented reality ArUco library developed by Rafael Muñoz and Sergio Garrido [RRMSMC18] [GJMSMCMC15].

The detection of ArUco markers is internally a two steps process. First, marker candidates are found on the image. This is achieved by adaptive thresholding followed by contour extraction. Only the convex and approximately square-like contours then advance further. Second, the internal structure of these candidates is examined to confirm they, in fact, are ArUco markers. After applying perspective transformation to get the markers into canonical form, the Otsu's method [Mor00] for thresholding is used to distinguish black and white bits of the marker. This image is subsequently divided into an array according to expected marker size. In each cell, the number of white pixels is compared to the number of black pixel to determine, whether it is a white or a black bit. Finally, the detection is considered successful, if the inner bit pattern is contained in the dictionary used. [mar15]

Similarly to chessboard calibration in the previous section, by obtaining correspondences between 3D points in the reference frame of a particular marker and 2D points on camera plane, marker pose with respect to camera can be estimated. Each marker has its own coordinate system attached to its center with the z-axis pointing up.

For the purpose of this project, multiple markers must be tracked at once (32

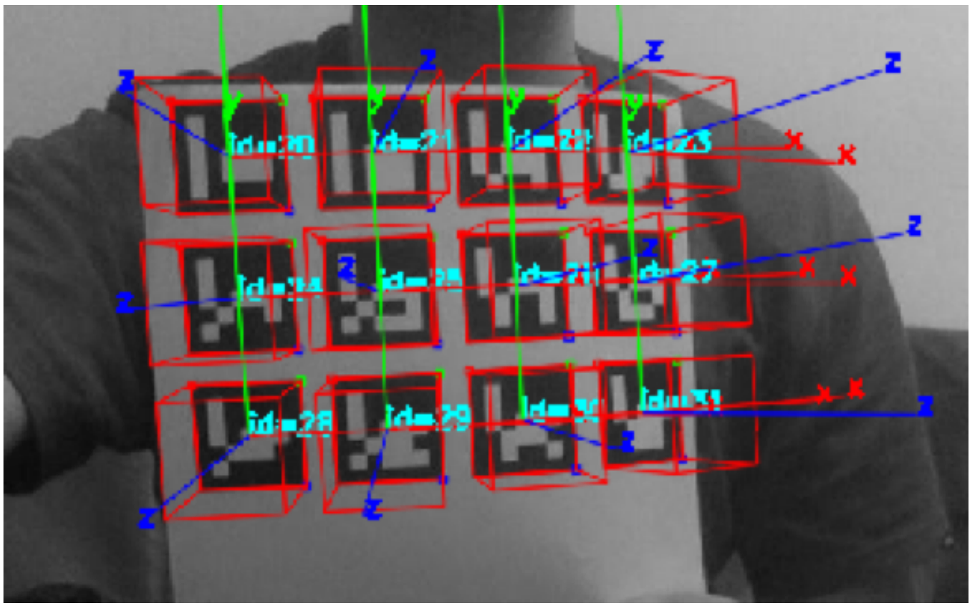


Figure 3.7: ArUco markers with drawn coordinate frames with respect to camera.

to be specific, one for each chess piece). To achieve this, `demo_aruco_markermap.launch` launch file of `tuw_marker_pose_estimation` package is used and modified a little to perform the detection on Intel RealSense camera data instead of the default camera (usually a webcam). This new launch file has been added to `tuw_marker_pose_estimation` package. [Bad18a] The steps necessary to turn the marker detection on are as follows:

1. Connect the RealSense camera via USB port, make sure the port is enabled if working on a virtual machine.

2. Turn the RealSense camera on through ROS:

```
$ roslaunch realsense2_camera rs_camera.launch
```

3. In a seperate terminal run the new launch file:

```
$ roslaunch tuw_marker_pose_estimation
    demo_aruco_markermap_realsense.launch
```

ArUco markers with ids 0-31 from the Original ArUco dictionary were used with 3.8x3.8 cm dimensions. To be detected, they each need to have a white padding - 0.15 cm on every side in this case.

Language control

The software used for language in this project was Python SpeechRecognition library. It supports a wide range of engines like CMUSphinx (offline), Google Speech Recognition, Microsoft Bing Voice Recognition or IBM Speech to Text. Example of usage for Google Speech Speech Recognition (utilized by `language_ctrl` package) is given here:

Listing 3.5: Google Speech Recognition example

```
import speech_recognition as sr

rec = sr.Recognizer()

with sr.Microphone() as source:
    print("Speak:")
    rec.adjust_for_ambient_noise(source)
    audio = rec.listen(source)
try:
    recognized = rec.recognize_google(audio, language="en-UK")
except sr.UnknownValueError:
    print("The audio was not understood.")
except sr.RequestError as e:
    print("Results could not be requested
    from service; {0}".format(e))
```

The components of a conventional automatic speech recognition (ASR) system and their connection is depicted in Figure 3.8.

In a nutshell, physical sound wave is transformed to an electrical signal by the microphone and then is discretized and quantized by the AD converter. The Acoustic analysis block in Figure 3.8 slices the received digital signal into 10 ms - 25 ms chunks known as speech frames. The idea behind this is, that speech viewed on a short timescale like this can be approximated as a stationary process, i.e. process whose statistical properties do not change over time. The signal is then used to extract acoustic features - these are vectors of real numbers that are meant to represent all the information in the signal frame. These features are usually Mel-frequency cepstral coefficients (MFCCs) [Lyo12] of the signal derived commonly like this:

1. Get power spectrum of the signal:

$$P(\omega) = |\mathcal{F}\{f(t)\}|^2, \tag{3.7}$$

   where f(t) is the input discrete signal frame and ω is the frequency.

2. Map the power spectrum to mel scale - a sound pitch scale that is based on a listener's perception of which tones are equidistant from one another rather than their actual frequency.

$$P'(\omega_{mel}) = M(P(\omega)), \tag{3.8}$$

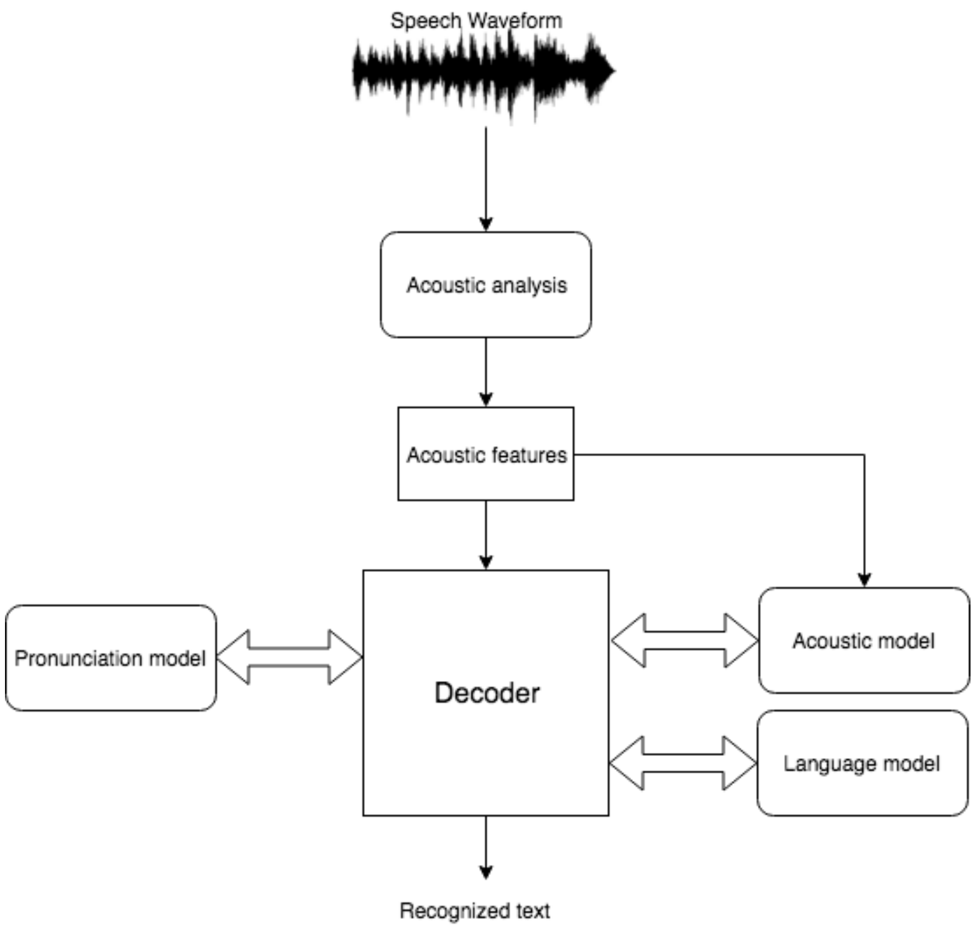


Figure 3.8: ASR system diagram [Res17], Speech Waveform image from [speer], drawn using https://www.draw.io/.

where M is conversion function.

3. Take the discrete cosine transform of the logarithm of this spectrum:

$$MFCC(k) = C(log(P'(\omega_{mel}))), \tag{3.9}$$

where C is the conversion function and $MFCC(k)$ is the final feature vector.

Now transitioning to the Acoustic model in Figure 3.8, it is used to give the Decoder probabilities of each possible sequence of phonemes the examined utterance could be consisting of. Phonemes are sound units, that distinguish one word from another in a particular language. Most languages have between 20 and 60 different phonemes. The input to the Acoustic model are the Acoustic features. On the inside, it consists of a Hidden Markov Model (HMM), Deep Neural Network (DNN) or a mixture of both. An HMM is essentially a statistical model of phonemes, in this case, as the hidden states that have some transition probabilities, i.e. probabilities to transition to some other hidden state, some emission probabilities, i.e. probabilities of "emitting" particular output - feature vector in our case, and finally there are initial state probabilities. These probabilities first have to be learned on training data consisting of feature vectors from a speech signal labeled with its phonemes. Figure 3.9 illustrates the HMM state diagram on an example of just five phonemes.

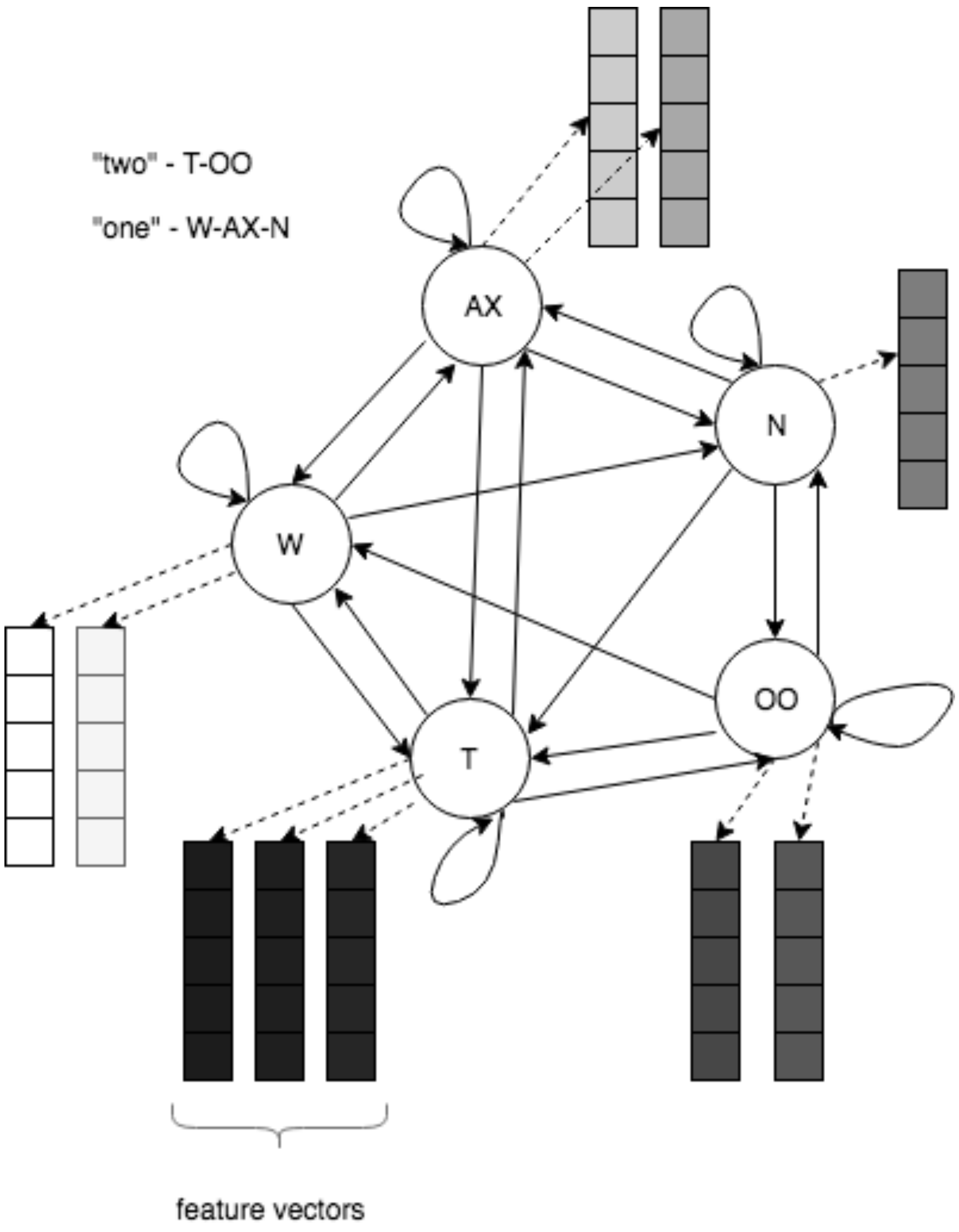


Figure 3.9: An example of HMM state diagram for acoustic model with five phonemes, drawn using https://www.draw.io/.

On the other hand, DNNs are based on multi-layer computational graphs, that try to improve their parameters, so called "weights", in order to turn their input into a desirable output. They are trained on the same data HMM would be and either output the recognized class - a particular phoneme, or determine the probabilities for an HMM.

After receiving probability distributions for all possible phoneme sequences, a way to transform a particular set of phonemes into words is needed. That is where the Pronunciation model comes in handy. Unlike the Acoustic model, it is quite straight-forward - dictionaries derived by linguistic experts with word to phonemes correspondences are utilized. Unfortunately, this textbook pronunciation can often vary from the real one in practice due to an accent or fast-paced speech.

Finally, the Language model provides probabilities of appearance of a particular word given a context, i.e. its N predecessors, so called Ngrams. To obtain these probabilities, large database of text is needed and for each word and Ngram pair, the probability is computed like so:

$$p(\text{"coffee"}|\text{"I need"}) = \frac{\pi(\text{"I need coffee"})}{\pi(\text{"I need"})}, \tag{3.10}$$

where the $\pi()$ function is number of occurrences of the string in the argument in the training set.

Although Recurrent Neural Networks (RNN) are used to solve this problem, too, Ngrams are still the faster solution. [Res17] [pytch]

KUKA robot

The robot used for this project was KUKA LBR Iiwa 7 depicted in Figure 3.10.

LBR means Leichtbauroboter - light construction robot, Iiwa stands for Intelligent industrial work assistant. The maximum mass of a load is 7 kg. This robot has seven rotational joints, therefore seven degrees of freedom. Basic specifications of each joint can be viewed in Table 3.1. [Č8]

Joint	Value range[°]	Max torque [Nm]	Max velocity [°/s]
1	± 170	176	98
2	± 120	176	98
3	± 170	110	100
4	± 120	110	130
5	± 170	110	140
6	± 120	40	180
7	± 175	40	180

Table 3.1: Basic joint specifications

Figure 3.10: KUKA LBR Iiwa 7, from [kuk19].

Joint	Joint type	θ [rad]	d [m]	a [m]	α [rad]
1	Rotational	φ_1	0.34	0	-$\pi/2$
2	Rotational	φ_2	0	0	$\pi/2$
3	Rotational	φ_3	0.4	0	$\pi/2$
4	Rotational	φ_4	0	0	-$\pi/2$
5	Rotational	φ_5	0.4	0	-$\pi/2$
6	Rotational	φ_6	0	0	$\pi/2$
7	Rotational	φ_7	0.126	0	0

Table 3.2: Denavit-Hartenberg parameters

This robot has open-loop kinematic chain, that can be described by Denavit-Hartenberg notation provided by Table 3.2.[Č8] The angles φ_i, $i = 1, ..., 7$ are the respective joint angles. Each row of this table provides information on how a reference frame connected with the previous joint transformed into its successor. To be more specific, the meaning of the last four columns is rotation around z-axis, translation along z-axis, translation along x-axis and rotation around x-axis respectively. The transformation matrix transforming points from the reference frame of joint i to that of joint (i-1) can be derived as follows:

$$T_i^{i-1} = \begin{bmatrix} cos(\varphi_i) & -sin(\varphi_i) & 0 & 0 \\ sin(\varphi_i) & cos(\varphi_i) & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha_i) & -sin(\alpha_i) & 0 \\ 0 & sin(\alpha_i) & cos(\alpha_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \quad (3.11)$$

$$\begin{bmatrix} cos(\varphi_i) & -sin(\varphi_i)cos(\alpha_i) & sin(\varphi_i)sin(\alpha_i) & 0 \\ sin(\varphi_i) & cos(\varphi_i)cos(\alpha_i) & -cos(\varphi_i)sin(\alpha_i) & 0 \\ 0 & sin(\alpha_i) & cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.12}$$

where d_i and α_i are DH parameters for joint i from Table 3.2. The final transformation matrix from the final joint to robot frame coordinates is:

$$T_7^0 = T_1^0 T_2^1 T_3^2 T_4^3 T_5^4 T_6^5 T_7^6. \tag{3.13}$$

The working envelope of this robot is depicted in Figures 3.11 and 3.12.

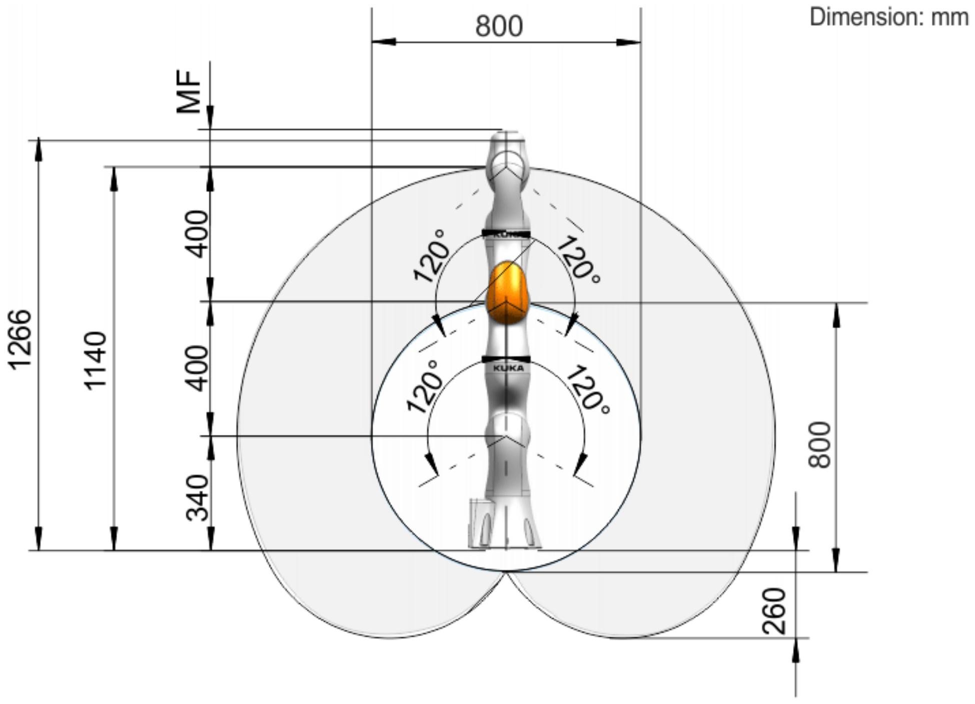


Figure 3.11: KUKA LBR Iiwa 7 working envelope side view, from [kuk16].

For this application, horizontal cuts through this envelope were important. These horizontal cuts form concentric circles. Their radii are dependent on the height above the table. Using Pythagorean theorem:

$$\begin{aligned} r_1(h) = r_1 &= \sqrt{R_1^2 - (R_1 - h - o)^2}, \quad h \in \langle 0, 740 \rangle mm \\ r_2(h) = r_2 &= \sqrt{R_2^2 - (R_2 - h - o)^2}, \quad h \in \langle 0, 1140 \rangle mm, \end{aligned} \tag{3.14}$$

where R_1 and R_2 are inner and outer sphere radii respectively, o is vertical offset of the spheres from the table (i.e. $o = 60$), h is the height above the table and r_1 and r_2 are inner and outer circles radii at that height.

The heights we are interested in for this project are $h_1 = 256mm$ (20 + 110 + 126 = piece height + length of gripper + last link length), i.e. in contact with a piece and $h_2 = 376mm$ (256 + 100 + 20 = h_1 + 100 + piece height)

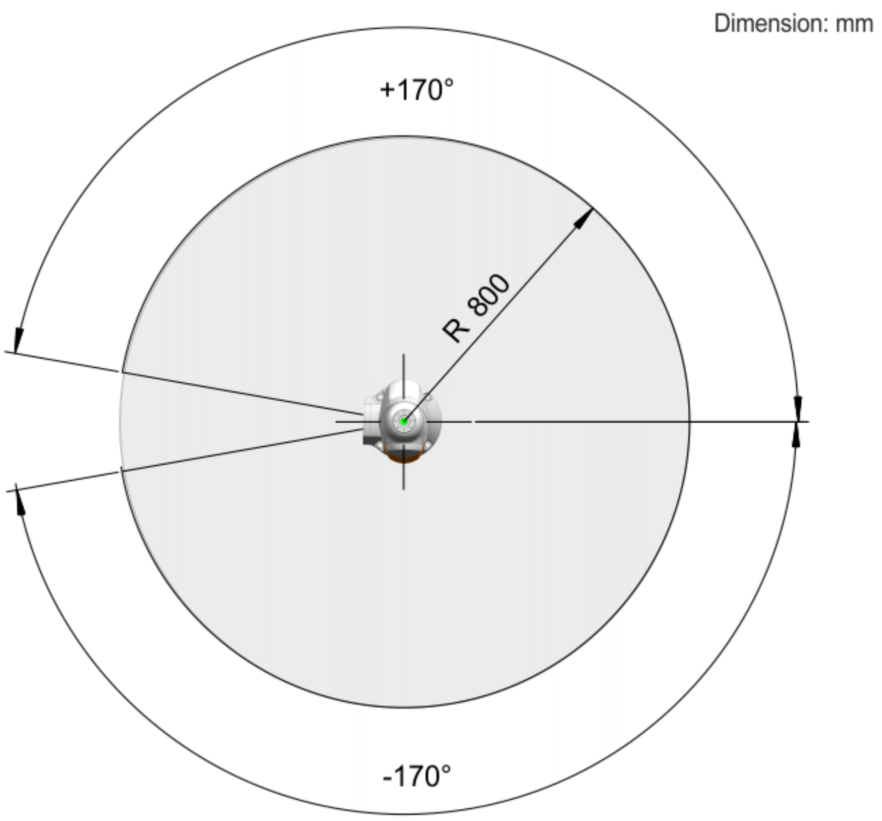


Figure 3.12: KUKA LBR Iiwa 7 working envelope top view, from [kuk16] .

above target piece position. Using 3.14:

$$
\begin{aligned}
r_1(h_1) &= 391.08mm \\
r_2(h_1) &= 795.58mm \\
\\
r_1(h_2) &= 398.38mm \\
r_2(h_2) &= 799.19mm.
\end{aligned}
\tag{3.15}
$$

The space the robot can effectively move chess pieces in this way is therefore restricted on the table by concentric circles with radii:

$$r_{in} = max\{r_1(h_1), r_1(h_2)\} = 398.38mm \tag{3.16}$$

and

$$r_{out} = min\{r_2(h_1), r_2(h_2)\} = 795.58mm. \tag{3.17}$$

The biggest square, that can be fitted between these two circles must have these two properties:

1. one of its sides is tangent to the inner circle
2. it has two corners on the outer circle .

The side of this square can be calculated using Pythagorean theorem on a right triangle as illustrated on 3.13. More specifically:

$$r_{out}^2 = (r_{in} + a)^2 + (a/2)^2, \tag{3.18}$$

where a is the side of the square. Solving this quadratic equation for a yields:

$$a = \frac{-4r_{in} + 2\sqrt{5r_{out}^2 - r_{in}^2}}{5}. \tag{3.19}$$

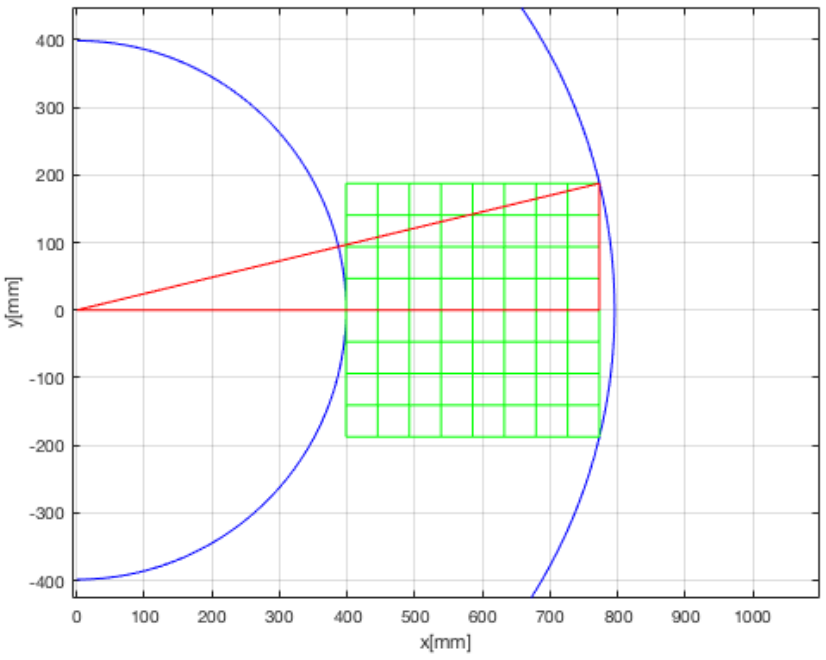


Figure 3.13: Effective working space (blue), the biggest chessboard, that can fit in (green) and a right triangle to compute its side (red), using MATLAB plot.

Substituting for r_{in} and r_{out}:

$$a = 374.8124mm. \tag{3.20}$$

In reality, the robot does not need to reach all the outer corners of the chessboard - the center of the outermost field is sufficient. Therefore, the final chessboard is bigger:

$$a_{fin} = 390mm. \tag{3.21}$$

The x coordinate of its center location is $c_x = 600mm$. This is illustrated in Figure 3.14.

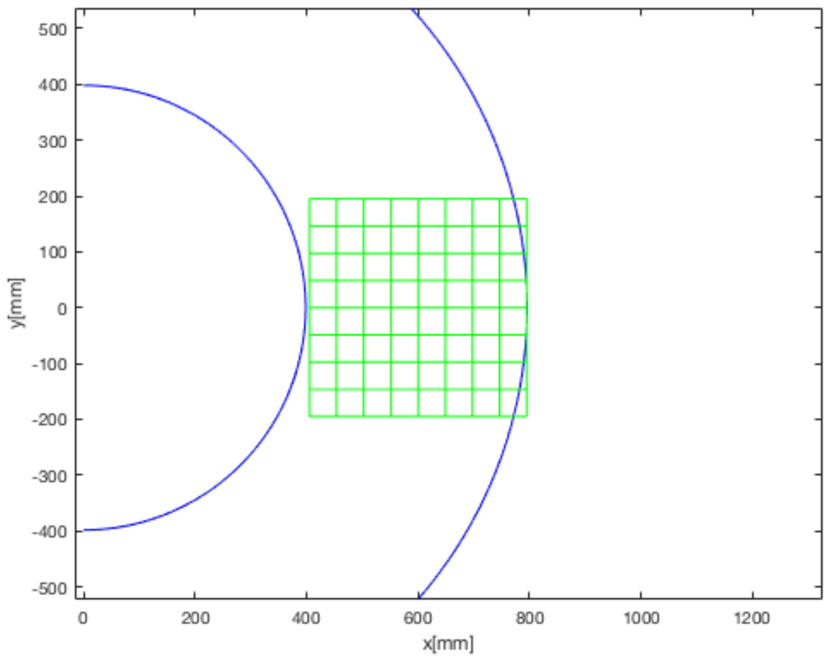


Figure 3.14: Effective working space (blue), the actual chessboard (green), using MATLAB plot.

Electromagnetic gripper

Steel plates on the upper side of the chess pieces (under markers) were picked up by an electromagnetic gripper, that was custom made for this application. The gripper in Figure 3.15 was 3D printed and is retractable for robot safety reasons. A 24 V electromagnet with 30 mm diameter depicted in Figure 3.16 resides in this gripper.

The electromagnet is controlled via ROS service call by writing to the particular robot end-effector output ('OutputX3Pin1'). To indicate the magnet is on, a green LED on robot's last link is lit up.

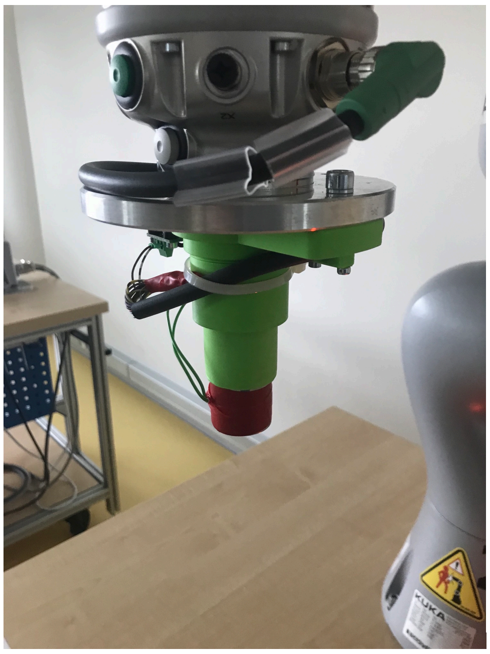


Figure 3.15: Electromagnetic gripper.

Figure 3.16: The electromagnet type used, from [con19].

Chapter 4
Results

Architecture description

The project main package is robot_chess_player. It takes inputs from tuw_marker_detection package and language_ctrl package and outputs control messages to robot using moveit_commander.

The language_ctrl package publishes on "/cmd" topic in the form of a string of recognized linguistic input. The tuw_marker_detection package on the other hand publishes on "/markersAruco" topic. The messages are of type Pose of geometry_msgs package - cartesian x,y and z coordinates in the camera coordinate system are provided as well as 3D rotation represented as a quaternion. This is common for any 3D configuration information in ROS.

This basic communication layout is presented in the block diagram in Figure 4.1.

The robot_chess_player package (available on the CD included with this thesis or on https://gitlab.ciirc.cvut.cz/jaluvmar/robot_chess_player) comprises of 7 nodes:

- language_interface.py
- chess_commander.py
- markers_spawner.cpp
- grabbed_piece_pose_publisher.py
- robot_grab.py
- robot_grab_moveit.py
- chess_gui.py .

The language_interface.py node subscribes to the "/cmd" topic, then parses the received string. If the incoming string is not one of the expected words (i.e. chessboard piece or target field), it notifies player with a voice error message. Upon reception of both valid piece type and target field in classic chessboard coordinates (e.g. 'A4'), language_interface.py publishes this information on "/chess_command" topic. It also subscribes to

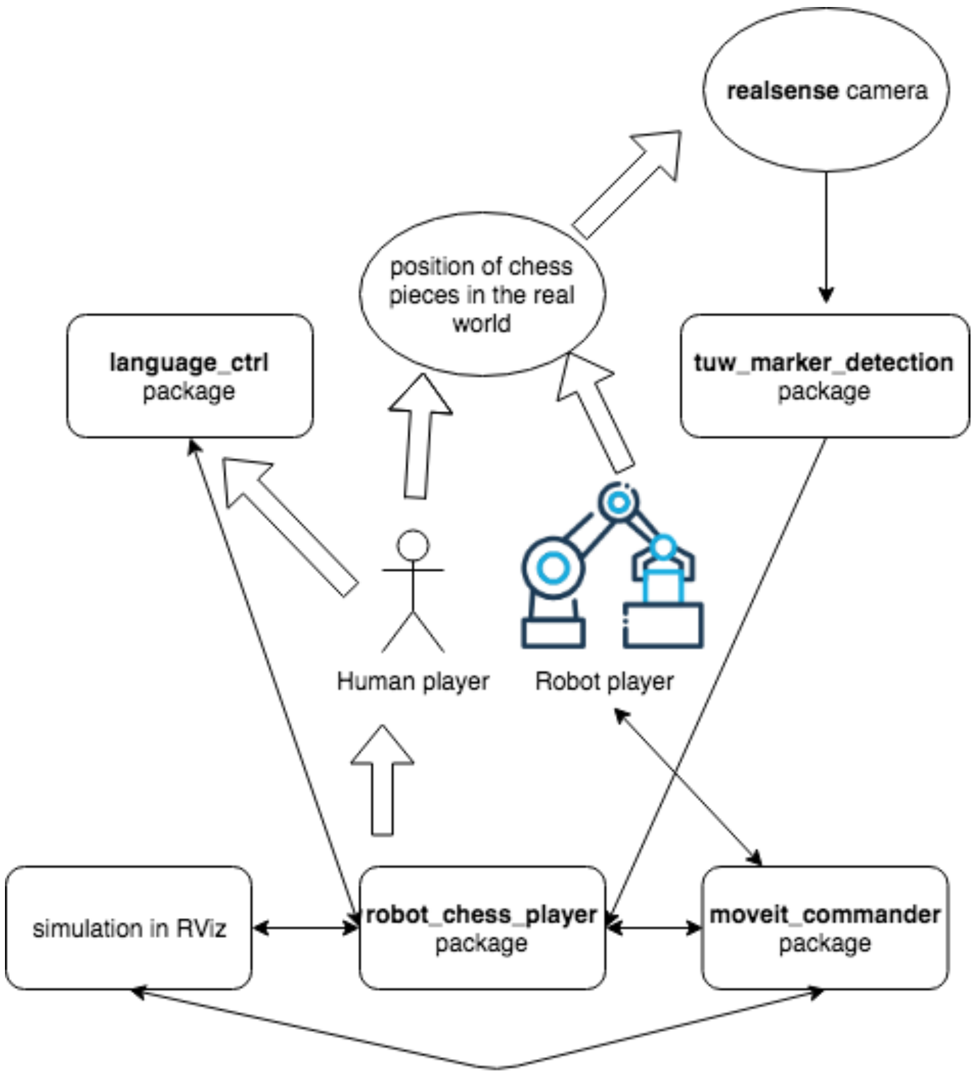


Figure 4.1: Block diagram of basic communication, drawn using https://www.draw.io/.

`"/speech_output_request"` topic and plays different voice messages based on received string.

The `chess_commander.py` node deals with chess game logic, which will be described later on, its communication interface however, is a part of this section. It receives chess piece move requests from the `language_interface.py` node via `"/chess_command"` topic, checks their validity in terms of game rules and reports any invalid moves back to `language_interface.py` node via `"/speech_output_request"`. Ambiguities and special situations are discussed with the player this way, too. When a move is valid, it is published on `"/requested_move"` topic as piece id and its desired position in numerical chessboard coordinates (e.g. m = 5, n = 1).

The `markers_spawner.cpp` is a central node, which provides essential functionality and is a backbone of the communication in this project. Firstly, it subscribes to "`/markersAruco`" topic, then uses accurate camera position and orientation parameters to transform ArUco markers' poses from camera coordinate frame to world. Knowing their poses in world coordinates and with known chessboard dimensions and placement, it is possible to determine chessboard configuration of individual chess pieces, i.e. integer chessboard coordinates. That information is published on "`/pieces_board_coords`" topic for `chess_commander.py` to keep track of the game configuration. The `markers_spawner.cpp` node also receives messages from `chess_commander.py` via "`/requested_move`" topic and being well aware of each piece's whereabouts along with the chessboard position and orientation, it translates piece id and target field chessboard coordinates into piece's and target's world coordinates. These are in the form of two `geometry_msgs` Poses published on `requested_move_pose` topic. Except when a particular ArUco marker is detected, the piece's coordinates in world frame can be altered when grabbed by the robotic arm - in this case, the piece is connected to the end-effector's frame and moves with it. For this purpose this node subscribes to "`/grabbed_piece_pose`" and "`/attach_piece`" topics. Finally, this node is responsible for advertising to "`/visualization_marker`" and "`/visualization_marker_array`" topics. These are used to place the chess pieces and the chessboard into simulation in RVIZ.

The `grabbed_piece_pose_publisher.py` is a simple node, that publishes pose of a piece when grabbed by the robotic arm. This pose is only the robot end-effector pose shifted in negative z direction by the length of gripper and half the piece height.

The `robot_grab.py` and `robot_grab_moveit.py` are essentially the same program. The only difference is that the first one uses the `capek_pycommander` package while the second one import from `moveit_commander` directly. These nodes subscribe to `requested_move_pose` topic and according to this received information command robot to move the particular piece from its pose to the target pose. Along with turning the electromagnet on using a service call, they let the other nodes know the piece has been attached by publishing on "`/attach_piece`" topic.

Finally, the `chess_gui.py` node displays current chessboard configuration graphically. [che] [CFVI19]

For more clarity on how the communication between nodes in this project works, `rqt_graph` in Figure 4.2 is provided as well.

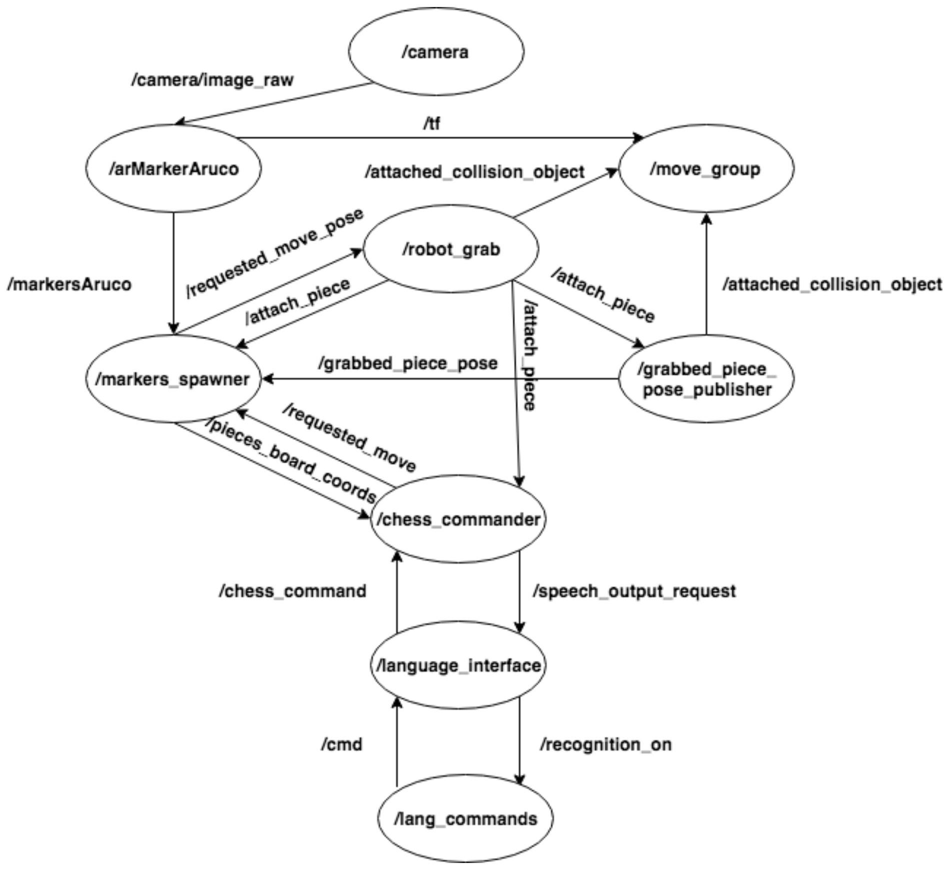


Figure 4.2: Project communication overview as `rqt_graph`, drawn using https://www.draw.io/.

Transformations between coordinate systems

The world coordinate system is placed exactly halfway between the two robots, while camera is recording the scene from above and its exact position in the world frame had to be computed by calibration. The Figure 4.3 shows RViz visualization of all essential coordinate frames used throughout this project. Except the two already mentioned, robot end-effector and chessboard center coordinate frames are included.

Positions and orientations of detected ArUco markers are provided by `tuw_marker_detection` package in camera coordinate frame, hence the need to transform to the world frame. Function `coords_transform` in `markers_spawner` node performs this transformation of all ArUco marker poses. As described in the OpenCv chapter, two different camera calibration methods were used, namely robot calibration witth ArUco marker and chessboard calibration. The output of the first method is camera position and quaternion orientation in the world frame, while chessboard calibration returns rotation vector and translation vector – $\boldsymbol{r}$ and $\boldsymbol{t}$ from now on – which transform points from world to camera coordinate frame. In each instance,

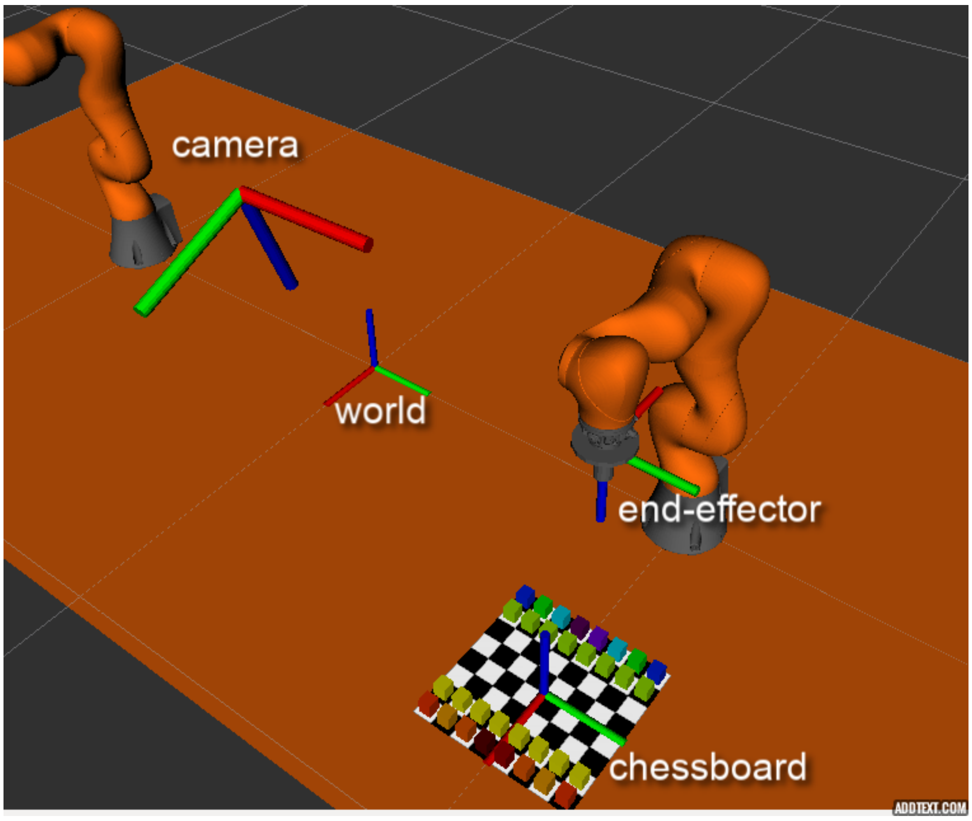


Figure 4.3: The essential coordinate frames used, text added using https://addtext.com.

different computation had to be carried out in order to obtain transformation matrix from camera to world frame. In the first case, transformation matrix is created as follows:

$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1-2(q_y^2+q_z^2) & 2(q_xq_y - q_zq_w) & 2(q_xq_z + q_yq_w) & 0 \\ 2(q_xq_y + q_zq_w) & 1-2(q_x^2+q_z^2) & 2(q_yq_z - q_xq_w) & 0 \\ 2(q_xq_z - q_yq_w) & 2(q_yq_z + q_xq_w) & 1-2(q_x^2+q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.1}$$

where x, y and z are coordinates of camera position vector in world frame and q_x, q_y, q_z and q_w are its orientation as normalized quaternions [Sho]. In the code, the toRotationMatrix() method from the Eigen library is used for the quaternions to rotation matrix conversion:

Listing 4.1: Transformation matrix creation

```
Eigen::Matrix3f R;
Eigen::Quaternionf q_in;
q_in.x() = CAMERA_Q_X;
q_in.y() = CAMERA_Q_Y;
q_in.z() = CAMERA_Q_Z;
q_in.w() = CAMERA_Q_W;
```

```
// normalization and conversion of quaternions
R = q_in.normalized().toRotationMatrix();

// converting rotation matrix to homogenous coordinates
Eigen::Matrix4f R_hom;
Eigen::MatrixXf v(1,3);
v << 0,0,0;
Eigen::MatrixXf w(4,1);
w << 0,0,0,1;
R_hom.block(0,0,3,3) = R;
R_hom.block(3,0,1,3) = v;
R_hom.block(0,3,4,1) = w;

Eigen::Matrix4f transl;
transl << 1,0,0,CAMERA_X,
      0,1,0,CAMERA_Y,
      0,0,1,CAMERA_Z,
      0,0,0,1;
Eigen::Matrix4f T;
T = transl*R_hom;
```

The $\boldsymbol{r}$ vector in the second case is an angle-axis rotation representation. The vector itself is a direction vector of the axis while its norm is the desired angle around this axis. This representation can be converted to rotation matrix. Let

$$\boldsymbol{u} = \frac{\boldsymbol{r}}{||\boldsymbol{r}||} = (x, y, z) \tag{4.2}$$

is $\boldsymbol{r}$ normalized. Also,

$$\varphi = ||\boldsymbol{r}|| \tag{4.3}$$

is its norm. Furthermore,

$$\begin{aligned} s &= sin(\varphi) \\ c &= cos(\varphi) \\ C &= 1 - c. \end{aligned} \tag{4.4}$$

The rotation matrix is then

$$R = \begin{bmatrix} x^2C + c & xyC - zs & xzC + ys \\ yxC + zs & y^2C + c & yzC - xs \\ zxC - ys & zyC + xs & z^2C + c \end{bmatrix} \text{[Bak17]}. \tag{4.5}$$

Using this rotation matrix and $\boldsymbol{t}$ to create transformation matrix in the same fashion as in 4.1 would have resulted in an incorrect outcome as $\boldsymbol{r}$ and $\boldsymbol{t}$ transform coordinates in the opposite way than needed. To get a correct transformation matrix, we need to use the opposite rotation (represented by

R^T) and opposite translation (-$\boldsymbol{t}$) in the switched order:

$$T = \begin{bmatrix} R^T & \mathbf{0} \\ 000 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\boldsymbol{t}_x \\ 0 & 1 & 0 & -\boldsymbol{t}_y \\ 0 & 0 & 1 & -\boldsymbol{t}_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.6}$$

where $\boldsymbol{t}_x$,$\boldsymbol{t}_y$ and $\boldsymbol{t}_z$ are x, y and z components of the translation vector.

After obtaining T either way, the position in the world frame of every detected piece is computed like this:

$$p_{h,w} = Tp_{h,c}, \tag{4.7}$$

where $p_{h,w}$ is position vector of particular ArUco marker in homogenous world coordinates and $p_{h,c}$ is the same vector in homogenous camera coordinates.

The orientation in the world frame is obtained by similar computation:

$$R_w = RR_c, \tag{4.8}$$

where R_w and R_c are rotation matricies of particular ArUco marker with respect to world and camera, respectively, and R is camera rotation matrix (before converting to homogenous coordinates). R_c is obtained from marker quaternion again by toRotationMatrix() method. R_w is then converted back to quaternion representation and along with position used to update global `pieces_config` array.

For most of the time, chess pieces are assumed to stand on the z = 0 plane in an upright position. Using this assumption, all markers must have fixed z position (the piece center - half the piece height for RViz markers) and only rotation around the z axis is allowed. The quaternion corresponding to this rotation has the following form:

$$\begin{aligned} q_x &= 0 \\ q_y &= 0 \\ q_z &= sin(\varphi/2) \\ q_w &= cos(\varphi/2), \end{aligned} \tag{4.9}$$

where φ is the angle around z axis. To figure out the quaternion in this form closest to the input quaternion, simple one-dimensional minimalization problem has been solved:

$$\begin{aligned} &min(||[0 \quad 0 \quad sin(\varphi/2) \quad cos(\varphi/2)] - [qar_x \quad qar_y \quad qar_z \quad qar_w]||) \\ &= min(\sqrt{qar_x^2 + qar_y^2 + (sin(\varphi/2) - qar_z)^2 + (cos(\varphi/2) - qar_w)^2}) \\ &= min(f(\varphi)), \end{aligned} \tag{4.10}$$

where qar is a given quaternion in the world coordinates not in the 4.9 form. A stationary point of this function was discovered:

$$\begin{aligned} \frac{df}{d\varphi} &= \frac{(sin(\varphi/2) - qar_z)cos(\varphi/2) - (cos(\varphi/2) - qar_w)sin(\varphi/2)}{2\sqrt{qar_x^2 + qar_y^2 + (sin(\varphi/2) - qar_z)^2 + (cos(\varphi/2) - qar_w)^2}} = 0 \\ &\iff qar_w sin(\varphi/2) - qar_z cos(\varphi/2) = 0. \end{aligned} \tag{4.11}$$

Then from the last equation:

$$\begin{aligned} tan(\varphi/2) &= \frac{qar_z}{qar_w} \\ \varphi &= 2atan2(qar_z, qar_w), \end{aligned} \tag{4.12}$$

where atan2 was used to get the correct angle. By substituting this stationary point to the second order derivative of this function, it has been confirmed, that it indeed has a local (and global) minimum in this point (the second derivative is in the stationary point positive for all possible qar coordinates except the $qar_z = 0$, $qar_w = 0$ situation where atan2 is not defined).

Having the position of each ArUco marker and by that each chess piece represented in the world frame is sufficient for the pick and place task as the desired robot end-effector coordinate frame pose described in world coordinates is an input to the robot controller. However, for determining each piece's position with respect to the chessboard, the chessboard connected frame is needed. Chessboard position with respect to the world is given by its x and y coordinate angle φ between its and world coordinate frame's x-axis. Horizontal orientation and z = 0 is assumed. The transformation matricies between world and chessboard frames are these:

$$T_b^w = \begin{bmatrix} 1 & 0 & 0 & c_x \\ 0 & 1 & 0 & c_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos(\varphi) & -sin(\varphi) & 0 & 0 \\ sin(\varphi) & cos(\varphi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.13}$$

$$T_w^b = \begin{bmatrix} cos(-\varphi) & -sin(-\varphi) & 0 & 0 \\ sin(-\varphi) & cos(-\varphi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.14}$$

where T_b^w transforms points from chessboard coordinate frame to the world and T_w^b the other way. c_x and c_y are chessboard center coordinates in world frame and φ is the angle already mentioned above.

The rest of the transformations described in this section are two-dimensional and only take place within the chessboard (or close to it, i.e. taken pieces fields) on z = 0 plane. These include a coordinate system connected with the upper left corner of the chessboard (from the human player's perspective), discrete chess field coordinate system with an origin in the same corner, m-n-coordinates for now on, and finally classical alphanumeric discrete coordinates. These are depicted in Figure 4.4.

Function `chessboard_ids_from_config` in `markers_spawner` assigns each piece a chess field it is standing on (i.e. the discrete coordinates written in white on 4.4). It cycles through all pieces and for each one it transforms its x and y in world to chessboard center coordinates (using 4.14 transformation matrix) and then goes through all the rows and columns of the chessboard to contain the piece in one of the fields. In fact, the algorithm looks three pieces beyond the chessboard on every side as some taken pieces might be there. For more clarity, the source code of this operation is provided:

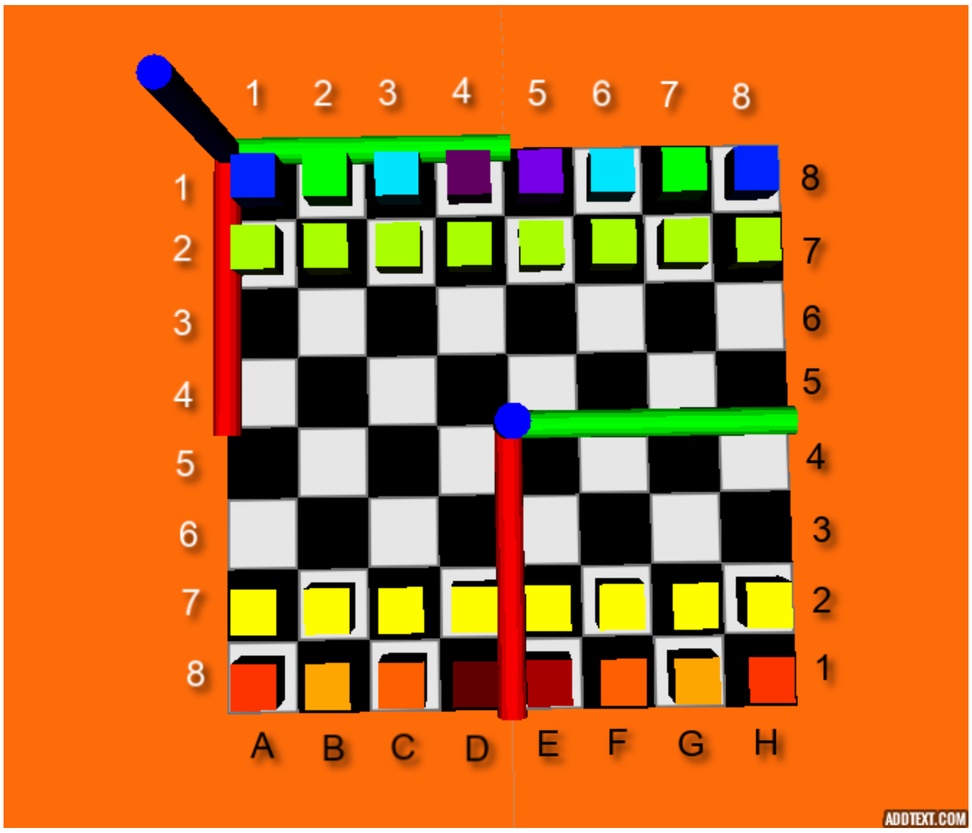


Figure 4.4: The chessboard coordinate systems, text added using https://addtext.com.

Listing 4.2: 3D position to discrete chessboard coordinates

```
for (int i =0; i<32; i++)
    {
        // piece pose in world frame
        config = pieces_config[i];
        p_hom << config.pose_x, config.pose_y, 0, 1;

        // transforming to chessboard center frame
        r_hom = T*p_hom;
        x = r_hom(0,0);
        y = r_hom(1,0);
        // starting at upper left corner
        //as if the chessboard was 11x11
        lower_bound = -CHESSBOARD_SIZE/2-3*CHESSBOARD_SIZE/8;
        for(int j =-2 ; j<12; j++)
        {
            upper_bound = lower_bound + CHESSBOARD_SIZE/8;
            if(x >= lower_bound && x < upper_bound)
            {
                board_coords[i].x = j;
            }
            if(y >=lower_bound && y < upper_bound)
            {
                board_coords[i].y = j;
            }
            lower_bound = upper_bound;
        }

        // check if piece is out of the chessboard
        //in an unexpected position

        if(board_coords[i].x < 1 || board_coords[i].x > 8)
        {
            board_coords[i].x = 0;
            board_coords[i].y = 0;
        }

        // check if piece is in taken pieces place and if so,
        // then convert to taken piece numbering

        else if(board_coords[i].y < 1 || board_coords[i].y > 8)
        {
            board_coords[i] =
            mn_to_taken_pieces_numbering(board_coords[i].x,
            board_coords[i].y);
        }

        // add to array for publishing

        chess_pose.id = i;
        chess_pose.m = board_coords[i].x;
        chess_pose.n = board_coords[i].y;
        msg.poses.push_back(chess_pose);

    }
```

The taken pieces are put on special fields dedicated to them on the sides of the chessboard. The black pieces (the ones closer to the robot) are put away to the left of the chessboard from human player's perspective and the white pieces to the right. They are organized to two columns of 6 and one column of 4, this is depicted in Figure 4.5 in simulation and a photo in Figure 4.6.

For easier manipulation with locations of these fields in the `chess_commander.py` node, separate numbering system is used for them. Black taken pieces locations are labeled $(-1, -1)$ to $(-16, -16)$ while the locations reserved for their white counterparts are labeled $(-17, -17)$ to $(-32, -32)$ as depicted in Figure 4.5.

While the code above finds chessboard coordinates for pieces given their

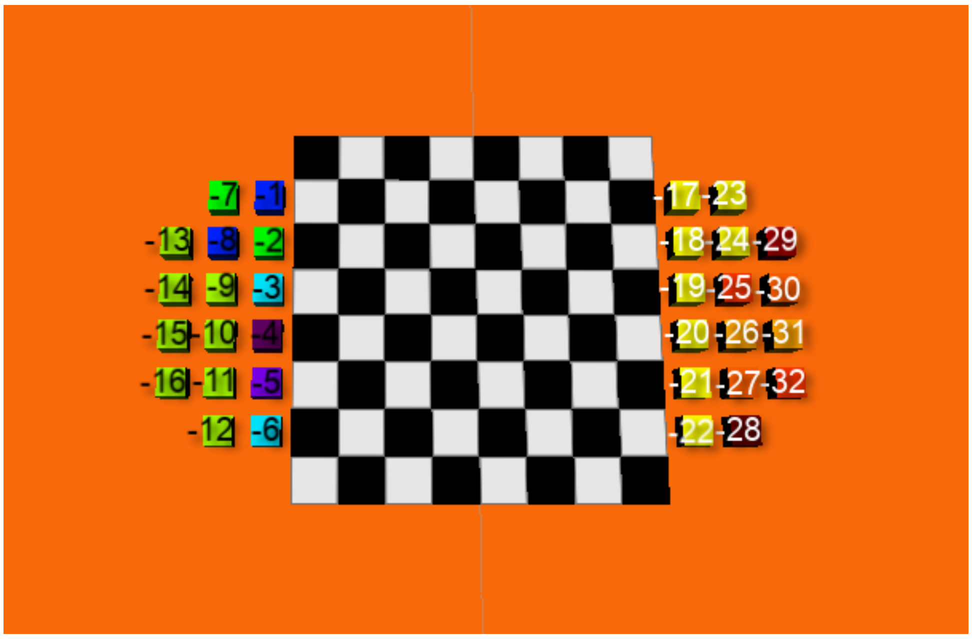


Figure 4.5: The taken pieces placement in the simulation, text added using https://addtext.com.

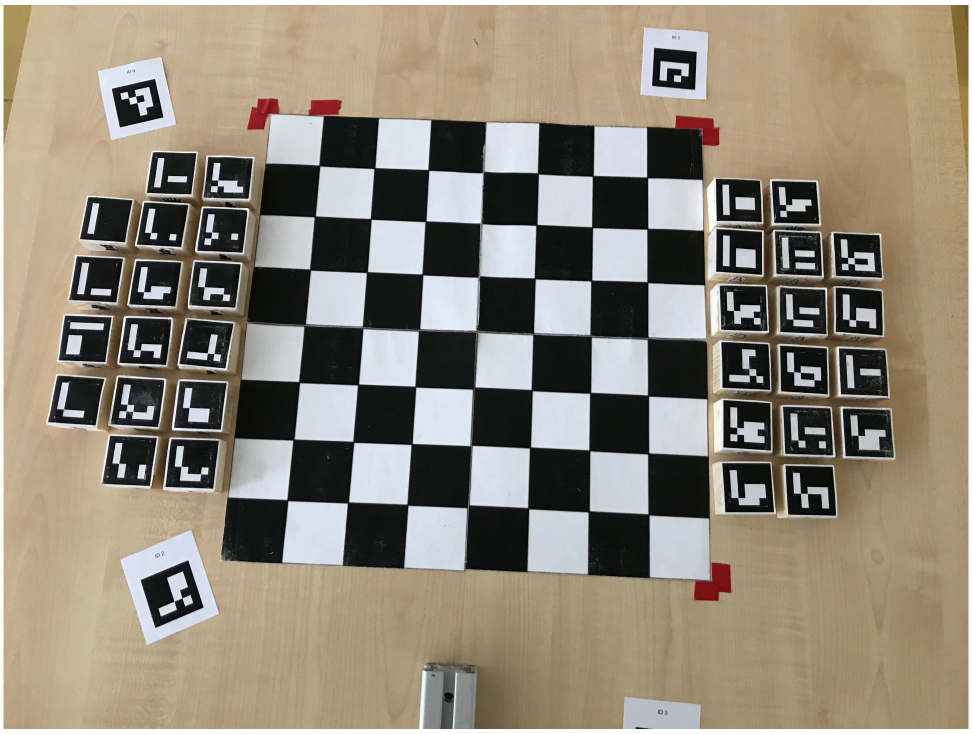


Figure 4.6: The taken pieces placement in the real setup.

world coordinates, the opposite transformation is also needed, i.e. when `chess_commander.py` demands a piece to move to a certain chess field, `markers_spawner.cpp` is required to provide `robot_grab.py` with the world coordinates of the center of that field. Given some numeric chessboard coordinates m, n, the x_{crn} and y_{crn} coordinates of the center of this field with respect to the chessboard corner frame is computed as:

$$\begin{aligned} x_{crn} &= (2m-1)\texttt{board\_size}/16 \\ y_{crn} &= (2n-1)\texttt{board\_size}/16. \end{aligned} \tag{4.15}$$

From there to the chessboard center frame, mere translation is performed:

$$\begin{aligned} x_{ctr} &= x_{crn} - \texttt{board\_size}/2 \\ y_{ctr} &= y_{crn} - \texttt{board\_size}/2. \end{aligned} \tag{4.16}$$

Lastly, translation and rotation obtained by 4.13 matrix is used to get to world coordinates. The function that computes this is `chessboard_coords_to_world`:

Listing 4.3: Chessboard to world coordinates conversion function

```
spatial_configuration chessboard_coords_to_world(
chessboard_coords config)
{
    int m, n;
    float x3,y3,x2,y2,x,y;
    spatial_configuration ret;
    chessboard_coords tmp;

    // first convert back from taken pieces numbering
    // if needed

    if(config.x < 0)
    {
        tmp = taken_pieces_numbering_to_mn(config.x);
        m = tmp.x;
        n = tmp.y;
    }
    else
    {
        m = config.x;
        n = config.y;
    }

    // discrete to corner chessboard coordinates
    x3 = (2*m-1)*CHESSBOARD_SIZE/16;
    y3 = (2*n-1)*CHESSBOARD_SIZE/16;
    // corner to center
    x2 = x3-CHESSBOARD_SIZE/2;
    y2 = y3-CHESSBOARD_SIZE/2;
    // center to world
    x = x2*cos(CHESSBOARD_PHI)-y2*sin(CHESSBOARD_PHI)+
    CHESSBOARD_X_POS;
    y = x2*sin(CHESSBOARD_PHI)+y2*cos(CHESSBOARD_PHI)+
    CHESSBOARD_Y_POS;

    ret.pose_x = x;
    ret.pose_y = y;
    ret.pose_z = 0;
    ret.quat_x = 0;
    ret.quat_y = 0;
    ret.quat_z = sin(CHESSBOARD_PHI/2);
    ret.quat_w = cos(CHESSBOARD_PHI/2);

    return ret;
}
```

Chess logic

Once the position of pieces on the chessboard is resolved by `markers_spawner.cpp` node, it is up to `chess_commander.py` to deal with the game logic, more specifically, it has the following features:

- move validity check
  - piece on chessboard
  - target on chessboard
  - target different from current position
  - trying to take your own pieces
  - clear path between source and target (for some types)
  - moving in accordance with piece type
  - moving into or staying in check
  - special moves - en passant, castling
- validity check for human player
  - move extraction from two consecutive chess configurations
  - detection of performing multiple moves
  - detection of moving your opponent's pieces
- taking pieces
  - searching for empty spots next to chessboard
  - 2-stage taking piece process
- returning pieces to the game

    - type available for returning
    - 2-stage returning piece process

- type - target field to piece id - target field
    - finding id of the piece based on target field reachability
    - ambiguity resolution

Starting with move validity check, the essential function here is `is_move_valid`. Given current chessboard configuration and requested move, i.e. id of a piece and target field m-n-coordinates, it returns whether that move is valid with respect to chess rules and if not, provides a reason why. First, a few general validity checks are performed, such as the piece and the target m-n-coordinates being on the chessboard or "friendly fire" (trying to take your own pieces):

Listing 4.4: General invalid move detection

```
def is_move_valid(chess_poses, piece_id, target_m, target_n,
check_check_on, output_on, language_input):

    # getting position of the piece from current chess
    configuration

    piece_m = chess_poses[piece_id].m
    piece_n = chess_poses[piece_id].n

    # getting id of piece currently on the target field,
    -1 if field is empty

    on_target_piece_id = piece_on_field(chess_poses, target_m,
    target_n)

    if(piece_m < 1  ):
        if output_on:
            output("MOVE INVALID: Piece has been taken!",
            language_input)
        return False


    if target_m == piece_m and target_n == piece_n:
        if output_on:
            output("MOVE INVALID: Piece is on that field
            already!", language_input)
        return False

    if target_m < 1 or target_m > 8 or target_n < 1 or
    target_n > 8:
        if output_on:
            output("MOVE INVALID: Moving out of chessboard!",
            language_input)
        return False

    if on_target_piece_id != -1 and ((is_white(piece_id)
    and is_white(on_target_piece_id)) or (is_black(piece_id)
    and is_black(on_target_piece_id))):
        if output_on:
            output("MOVE INVALID: You cannot take your own
            pieces!", language_input)
        return False
```

After this, the function branches - the move must be valid for that specific piece type. Where needed, the path from source field to target field is checked for other pieces in the way by the following function:

Listing 4.5: Detection of a clear path between source and target

```
def path_is_clear(chess_poses, piece_m, piece_n, target_m,
target_n):
    incr_m = sign(target_m - piece_m)
    incr_n = sign(target_n - piece_n)
    checked_field_m = piece_m
    checked_field_n = piece_n
    dist = max(abs(target_m-piece_m), abs(target_n-piece_n))
    for i in range(1,dist):
        checked_field_m += incr_m
        checked_field_n += incr_n

        if piece_on_field(chess_poses, checked_field_m,
        checked_field_n) != -1:
            print("MOVE INVALID: There is a piece in the way!")
            return False
    return True


def sign(x):
    if x >0:
        return 1
    elif x == 0:
        return 0
    else:
        return -1
```

For all piece types, it is then also determined whether the requested move would get the friendly king to check. This is achieved by `is_field_safe` function, which calls `is_move_valid` to determined if any of the opponent's pieces would be able to reach king's position after playing the requested move. For this to make sense (and avoid infinite recursion), `is_move_valid` called from within `is_field_safe` must not check for king in check:

Listing 4.6: Detecting whether friendly king is in check after a move

```
def is_field_safe(chess_poses, piece_id ,target_m, target_n):
    piece_is_black = is_black(piece_id)
    target_chess_poses = copy.deepcopy(chess_poses)
    on_field_id = piece_on_field(target_chess_poses, target_m,
    target_n)

    # create target chess configuration

    # if there is an opponent's piece on the target field now,
    it is placed out of chessboard

    # id 3 - black king, id 27 - white king

    if(on_field_id!=-1):
        target_chess_poses[on_field_id].m = 0
        target_chess_poses[on_field_id].n = 0

    target_chess_poses[piece_id].m = target_m
    target_chess_poses[piece_id].n = target_n

    # determine whether an opponent's piece could reach king
    after this move

    if(piece_is_black):
        for i in range(16, 32):
            if(is_move_valid(target_chess_poses, i,
            target_chess_poses[3].m, target_chess_poses[3].n,
            False, False, False)):
                return False
        return True
    else:
        for i in range(0, 16):
            if(is_move_valid(target_chess_poses, i,
            target_chess_poses[27].m, target_chess_poses[27].n,
            False, False, False)):
                return False
        return True
```

Building on this, the `play_move` function provides basic chess playing features - given id of a piece and target field m-n-coordinates and after move validity check, it sends the requested move to `markers_spawner.cpp`, which in turn mobilizes the `robot_grab.py` to move the robot. If an opponent's piece needs to be taken out of the chessboard in order to realize this move, it is put on the first empty field reserved for taken pieces of this color and only after this is the original move performed. Returning pieces back to game upon reaching the other side of the chessboard with a pawn is resolved here as well.

Finally, to make the user interface more natural, the `play_move_type_field` function wrapped around `play_move` enables the requested moves to be in the form piece type - target field. When more pieces of the same type can reach the same field, this is ambiguous. For this reason, a list of possible piece ids is first created and if it has more than one member, the player is asked to clarify by providing the piece's source coordinates.

To keep everything fair, the move of the human player needs to be checked for validity as well. This is commanded move validity check extended by certain features needed when using detection as depicted in Figure 4.7 . This presents a little challenge, because the move needs to be first extracted

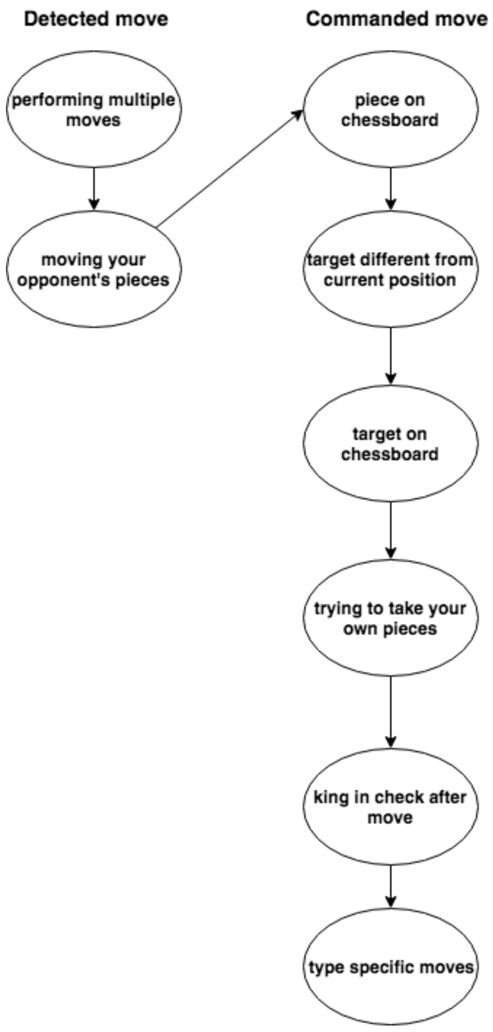

Detected move
Commanded move
performing multiple moves
moving your opponent's pieces
piece on chessboard
target different from current position
target on chessboard
trying to take your own pieces
king in check after move
type specific moves

Figure 4.7: Succession diagram of validity check components, drawn using https://www.draw.io/.

from the difference between chess configuration before and after the move. For most cases, checking if just one friendly piece was moved and then using `is_move_valid` would suffice, but this approach breaks down when playing moves involving taking opponent's pieces or returning pieces to the game. The following finite state machine in Figure 4.8 implemented in `compare_chess_poses` function solved this issue for returning pieces.

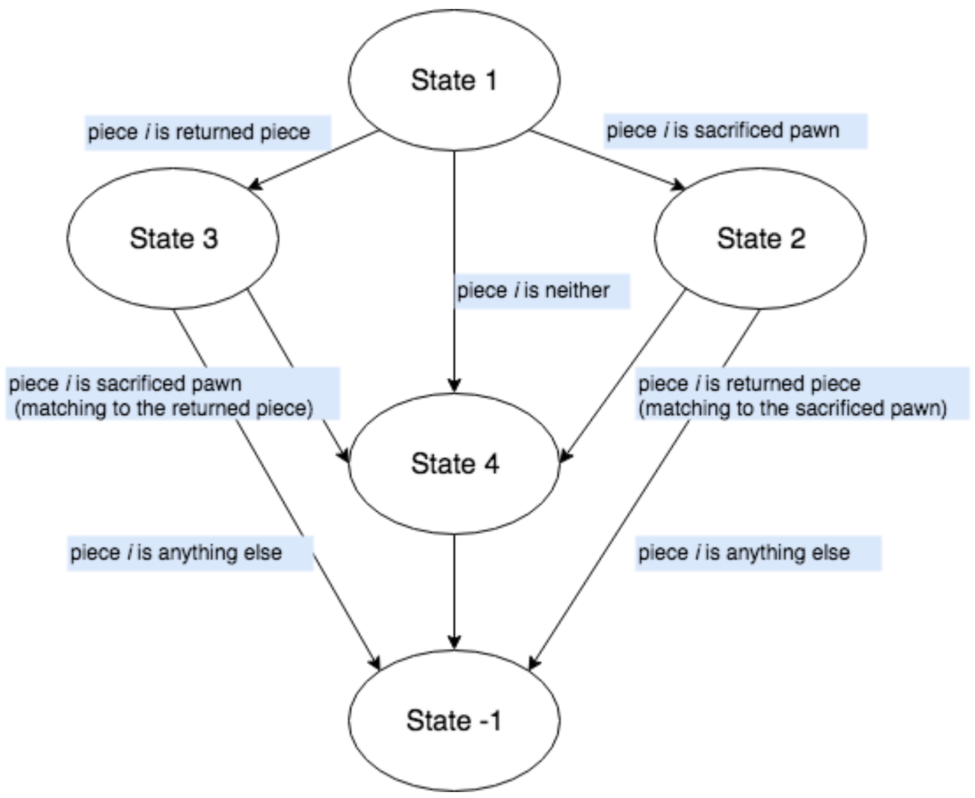

State 1
piece *i* is returned piece
piece *i* is sacrificed pawn
State 3
State 2
piece *i* is neither
piece *i* is sacrificed pawn (matching to the returned piece)
piece *i* is returned piece (matching to the sacrificed pawn)
State 4
piece *i* is anything else
piece *i* is anything else
State -1

Figure 4.8: The finite state machine for friendly chess piece i, which moved recently, drawn using https://www.draw.io/.

User interface

A two-player chess game was developed. The first player (usually playing as black) is meant to control the robot by voice to play for him, while the second player (usually playing as white) moves pieces the old fashion way. This game's main loop flowchart is in Figure 4.9.

Optionally, the game can be played using keyboard commands and text responses instead of language output and input. This is set through `chess_commander.py`'s boolean variable `language_input`. Similarly, the chess pieces belonging to the "human player" can be moved physically and the change detected or moved virtually based on user input. This can be switched off and on in `chess_commander.py` as well using `detecting_pieces` boolean variable. Lastly, switching sides is also possible via `playing_as_black` vari-

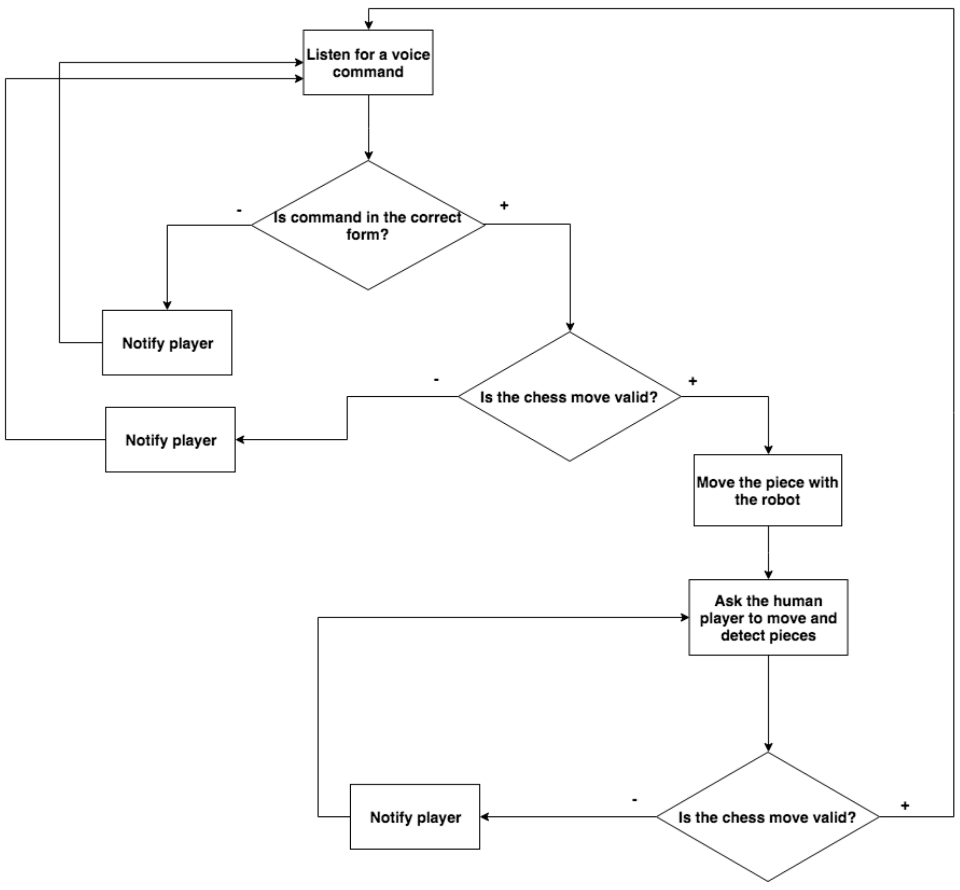

Listen for a voice command
Is command in the correct form?
-
+
Notify player
Is the chess move valid?
-
+
Notify player
Move the piece with the robot
Ask the human player to move and detect pieces
Notify player
-
Is the chess move valid?
+

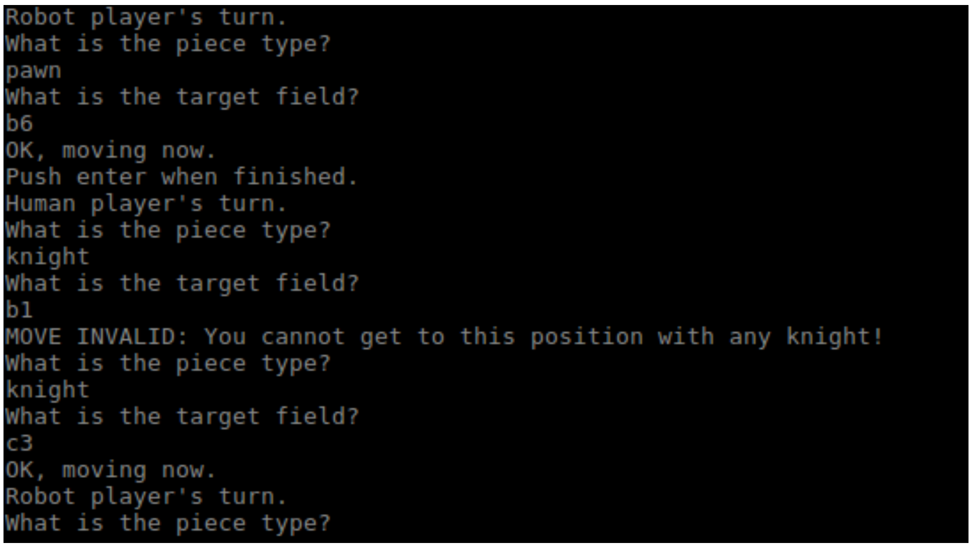
Figure 4.9: Chess game main loop, drawn using https://www.draw.io/.

able, in this case, the robot will move with the white pieces and the human with the black ones.

An example of this user/program interaction can be seen on the console output in Figure 4.10 (`language_input` and `detecting_pieces` variables are both set to False for this example). The corresponding chess configurations can be viewed in Figures 4.11, 4.12 and 4.13.

Error rate evaluation

To evaluate the computational expensiveness of each part of the architecture, time necessary for each key operation was measured (1.6 GHz Intel Core i5 used) as shown by Table 4.1. Rather than the absolute values measured, their ratio is more descriptive. The move validity check for pawns takes noticeably longer, because all 8 pawns need to be checked. Move component here refers to a part of the pick and place movement (e.g. getting above the pick position). The robot pick and place action includes planning, waiting and executing the whole action.



```
Robot player's turn.
What is the piece type?
pawn
What is the target field?
b6
OK, moving now.
Push enter when finished.
Human player's turn.
What is the piece type?
knight
What is the target field?
b1
MOVE INVALID: You cannot get to this position with any knight!
What is the piece type?
knight
What is the target field?
c3
OK, moving now.
Robot player's turn.
What is the piece type?
```

Figure 4.10: Console output example.

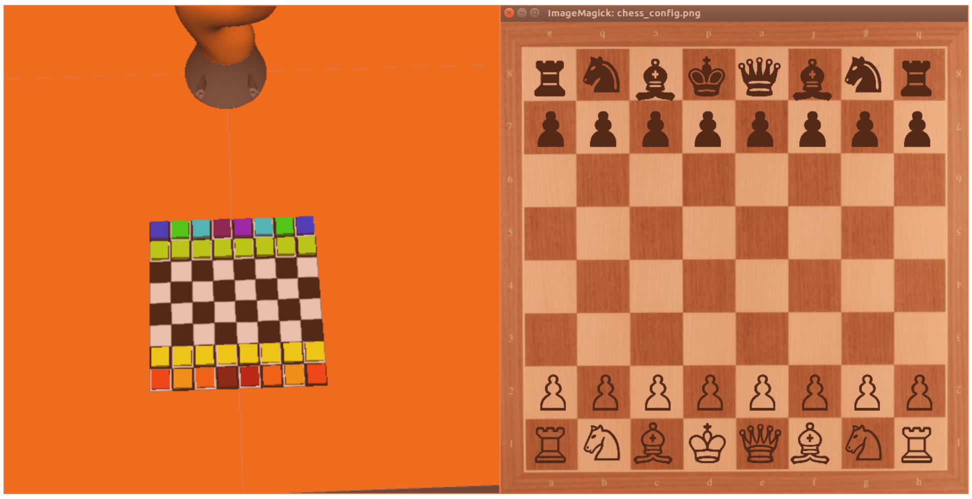


Figure 4.11: Initial configuration.

Component	Time took[ms]
ArUco markers detection	180
Validity check - pawns	30
Validity check - other pieces	1
Speech recognition from recording	1000
Robot move component planning	300
Robot move component execution	1000 - 2000
Robot pick and place action planning	2000
Robot pick and place action	30 000

Table 4.1: Execution time of architecture components.

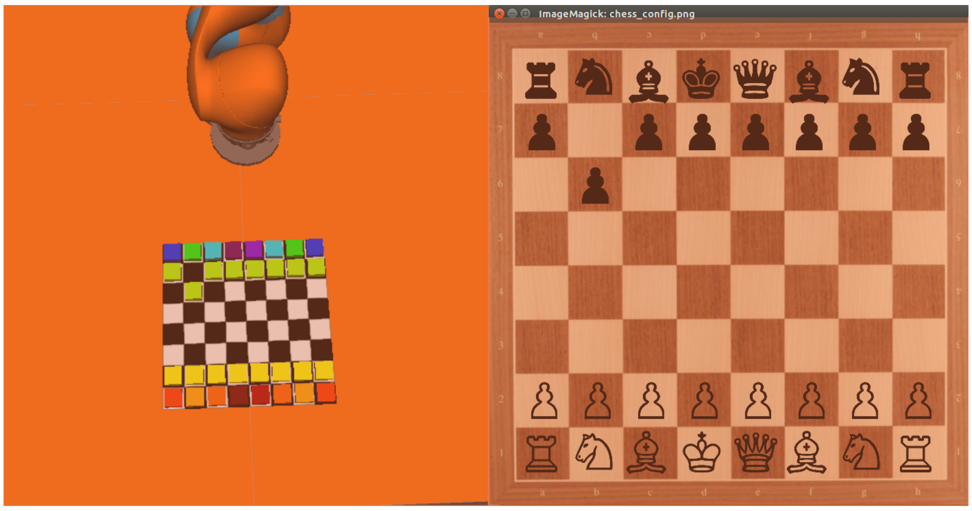


Figure 4.12: Configuration after instruction "pawn to b6".

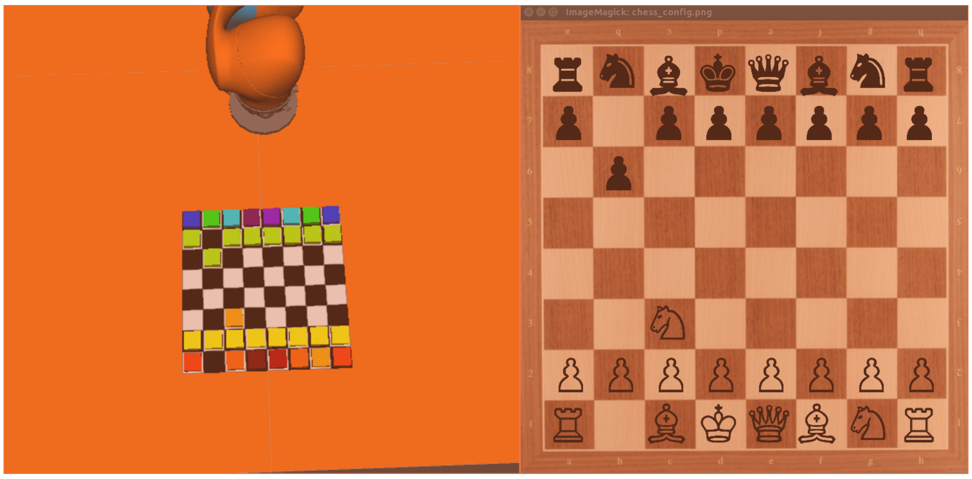


Figure 4.13: Configuration after instruction "knight to c3".

Different parts of this project were first tested separately, starting with robot accuracy. During this test, a piece was moved by the robot to a field surrounded by other pieces. To eliminate the inaccuracies of detection, the piece was picked up from the center of the field it was detected on, instead of its precise detected position. This approach proved so effective, it has stayed beyond the testing.

For the first round of testing, a configuration depicted in Figure 4.2 was constructed and the knight was moved by the robot from its initial position between the pawns and back. This was repeated in 5 cycles (i.e. 10 robot moves in total). This test was performed on 4 different locations on the chessboard and the results are summarized by Table 4.2.

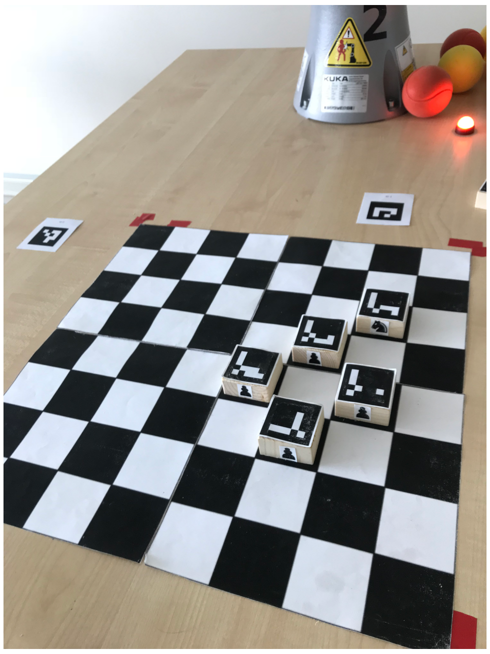


Figure 4.14: Experimental setup for robot accuracy test - moving knight piece between g6 and f4.

test number	initial/target location	cycles successful/performed
1	g6/f4	5/5
2	d3/b4	5/5
3	f2/h1	5/5
4	b7/d8	5/5

Table 4.2: A summary for robot accuracy tests – moving knight piece between two locations.

To make it more difficult for the robot and to make sure a significant error wouldn't accumulate over larger number of moves with one piece, a second test was devised. The configuration of this test is in Figure 4.15.

The knight was moved from g6 to f4, then to d3, c5, d7, f8 and back to g6. Three uninterrupted cycles, that is 18 moves, were performed with the robot. They all were successful as no collision between the knight and the other pieces occurred.

Next, piece detection accuracy was evaluated. The main problem with this is oscillation of a detected piece, that sometimes places the piece on a neighbouring field. To study this the `chess_gui.py` node was edited to not only create graphical output according to current chessboard configuration, but to count correct and incorrect configurations received based on a correct reference configuration.

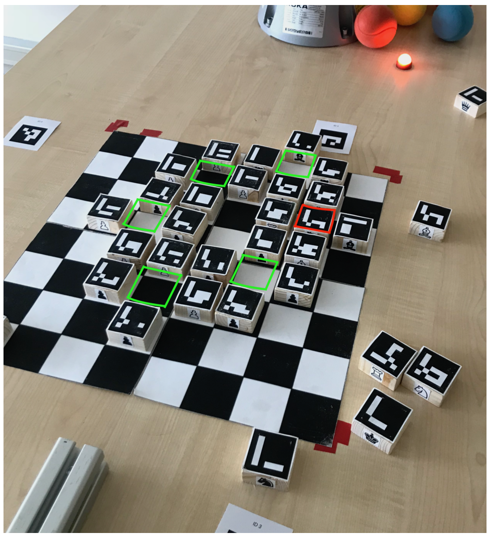


Figure 4.15: Experimental setup for robot accuracy test – moving knight piece around chessboard, knight initial position in red, way points in green.

The first configuration was initial chess configuration, five measurements were performed, each with 200 configuration updates. The time between consecutive updates was approximately 1s. The results are summarized in Table 4.3.

measurement number	correct configurations	incorrect configurations
1	173	27
2	179	21
3	154	46
4	158	42
5	143	57

Table 4.3: Detection accuracy test 1.

The next configuration for this detection test was devised from the previous by applying four chess moves. These chess moves along with detection results are depicted in the Tables 4.4, 4.5, 4.6, 4.7, 4.8 and 4.9.

round number	white	black
1	pawn g4	pawn b5
2	bishop g2	pawn c6

Table 4.4: Transition from configuration 1 to configuration 2.

measurement number	correct configurations	incorrect configurations
1	176	24
2	170	30
3	184	16
4	184	16
5	183	17

Table 4.5: Detection accuracy test 2.

round number	white	black
1	pawn f4	pawn g6
2	queen f2	pawn f5

Table 4.6: Transition from configuration 2 to configuration 3.

measurement number	correct configurations	incorrect configurations
1	200	0
2	200	0
3	200	0

Table 4.7: Detection accuracy test 3.

round number	white	black
1	pawn d4	knight f6
2	queen e3	rook g8

Table 4.8: Transition from configuration 3 to configuration 4.

measurement number	correct configurations	incorrect configurations
1	100	0
2	100	0
3	100	0

Table 4.9: Detection accuracy test 4.

During detection test 1 and 2, the erroneous configurations were detected less frequently than the correct ones, but more importantly were not in batches, but rather well distributed. This opens up the possibility to filter them out entirely by saving some small number of consecutive configuration updates (for example 10) and out of those declare the most frequent configuration within these updates to be the correct one. This is of course a trade-off - robustness for update time.

However, upon viewing the results of detection test 3 and 4, any incorrect configurations seem to disappear. At first this was attributed to the chess pieces being more spaced out and therefore better detectable, but then the initial configuration was measured again to test this hypothesis and it got similar results to tests on configurations 3 and 4. The only logical explanation here is that the scene illumination throughout the measurements improved due to the later experiments being carried out after 5 p.m. with the sun low enough to shine through the windows from the west. The chessboard itself was not directly illuminated by the sun, but overall light intensity must have improved.

The language was tested separately on three subjects. The tested subjects were asked to utter each piece type 10 times and eight target positions - A1 - A8 - were each pronounced 5 times. The A fields were chosen, because they are often mistaken for E fields. The results of this testing is in Tables 4.10, 4.11 and 4.12.

This score can be improved with a little practice - Table 4.13 depicts the author of this thesis attempting the same task. When performed by untrained subjects, of all words uttered 65% were correctly recognized, while the reference testing had 87% success rate.

word	# correct	# incorrect	# not heard
pawn	8	0	2
knight	2	7	1
bishop	10	0	0
rook	5	4	1
queen	7	2	1
king	1	9	0
A1	4	0	1
A2	4	0	1
A3	2	2	1
A4	2	2	1
A5	4	1	0
A6	3	1	1
A7	3	1	1
A8	2	3	0

Table 4.10: Language test, subject 1.

word	# correct	# incorrect	# not heard
pawn	2	8	0
knight	10	0	0
bishop	8	0	2
rook	9	0	1
queen	10	0	0
king	8	2	0
A1	5	0	0
A2	4	1	0
A3	2	3	0
A4	4	1	0
A5	4	1	0
A6	4	1	0
A7	4	1	0
A8	2	3	0

Table 4.11: Language test, subject 2.

word	# correct	# incorrect	# not heard
pawn	4	5	1
knight	6	3	1
bishop	9	0	1
rook	6	3	1
queen	7	3	0
king	2	8	0
A1	4	0	1
A2	4	1	0
A3	4	1	0
A4	4	1	0
A5	4	0	1
A6	3	2	0
A7	4	1	0
A8	1	4	0

Table 4.12: Language test, subject 3.

To conclude, a final test was conducted with setup depicted in Figure 3.1. Both players performed 20 chess moves, 10 each. As oscillation were dealt with by the method described above, the only problem was, that some markers were occasionally not detected at all. This leaves the program with their last location. This manifested as erroneous classification of the human player's move as invalid. In 4 cases out of these 20 moves, the `chess_commander.py` had to be restarted to allow taking sample of the correct chess configuration.

An example of markers not being detected is in Figure 4.16 . Despite the fact, that in this screenshot 11 markers were not recognized, most of them

word	# correct	# incorrect	# not heard
pawn	9	1	0
knight	10	0	0
bishop	10	0	0
rook	9	1	0
queen	10	0	0
king	5	5	0
A1	4	0	1
A2	4	1	0
A3	4	1	0
A4	4	1	0
A5	5	0	0
A6	4	1	0
A7	5	0	0
A8	4	1	0

Table 4.13: Reference testing.

were detected sporadically - at least once every few seconds. This is enough for this application to keep track of them. Up to two pieces were usually not detected stably - the piece on f3 in Figure 4.16 for example.

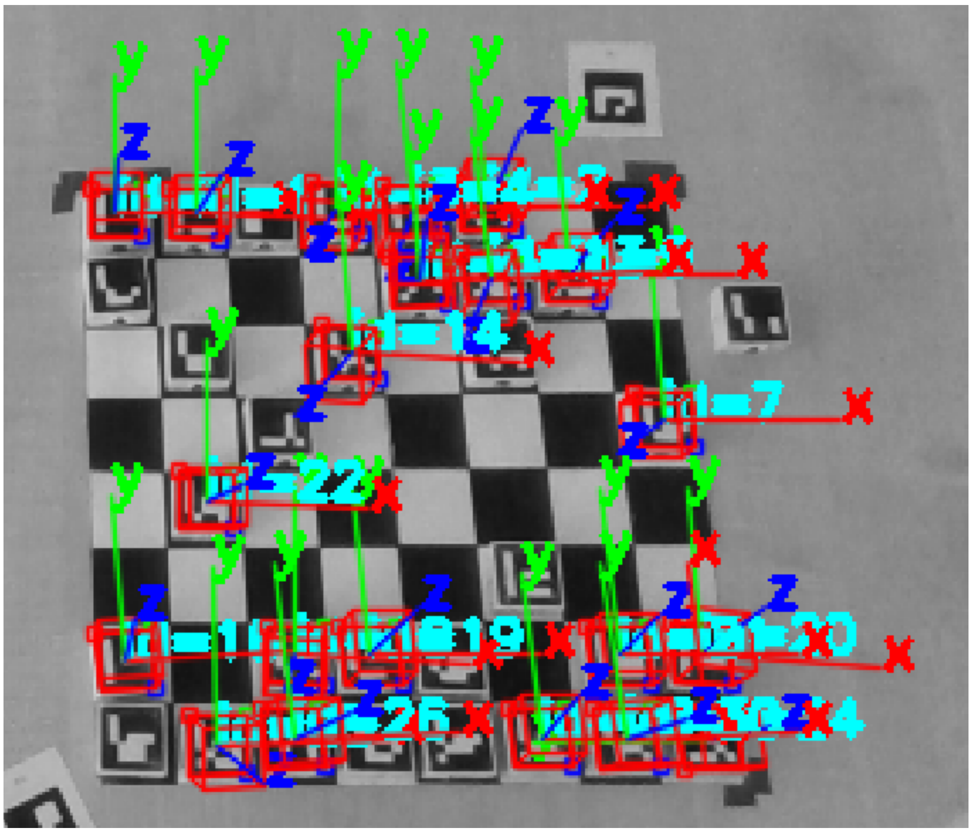


Figure 4.16: Detection of markers on the chessboard.

Chapter 5

Conclusion and Discussion

A framework for playing chess with KUKA robotic arm has been created. All technologies used have been described in the Chapter 3 including visual and voice input, robotic arm and custom built chess set. Chapter 4 described how exactly the technologies cooperated and communicated and it also provided little insight into the chess logic programmed specifically for this purpose. In the end, the whole system was evaluated.

Although the detection of chess pieces still has some issues and the human's move is in some cases falsely considered to be invalid as a result, overall the goal of this thesis has been achieved.

The detection problem may be solved by better choice of ratio between size and padding of the markers. For further improvement in accuracy, depth point cloud could be used as well. The other parts of this project work pretty well - the chess logic correctly responds to incoming chess configurations, the robot moves pieces precisely as demanded and the words from the small vocabulary used for this game are recognized well – 65% success rate of the untrained users can be easily improved by a little practice, proven by the 87% success rate of the reference testing.

Next steps from here could be for example usage of an open source automatic chess playing algorithm to either advise the player how to move next, or let the robot play on its own. In this case, the voice commands could be used to let the robot know when its turn began. Also, the chess pieces could be created to resemble the classical ones. Without any markers and relying only on recognizing their different shapes, a neural network would have to be trained. Ultimately, another similar board games like checkers could be added to robot's repertoire.

Appendix A

Bibliography

[AB16] H. Wörn A. Bihlmaier, *Hands-on Learning of ROS Using Common Hardware*, Robot Operating System (ROS). Studies in Computational Intelligence **625** (2016).

[ASHM15] Firas Al-Saedi and Ali H. Mohammed, *Design and Implementation of Chess-Playing Robotic System*, IJCSET **5** (2015), 90–98.

[Bad18a] M. Bader, *`tuw_marker_detection` package*, `https://github.com/tuw-robotics/tuw_marker_detection`, 2018, [Online; accessed May 21, 2019].

[Bad18b] Markus Bader, *`marker_msgs/MarkerDetection`*, `http://docs.ros.org/api/marker_msgs/html/msg/MarkerDetection.html`, 2018, [Online; accessed May 21, 2019].

[Bak17] M. J. Baker, *Maths - AxisAngle to Matrix*, `https://www.euclideanspace.com/maths/geometry/rotations/conversions/angleToMatrix/index.htm`, 2017, [Online; accessed May 21, 2019].

[BCW17] K. Merckaert D. Pinto M. van den Brande B. Convens, M. Lecomte and Q. Wets, *Chess Playing Robot*, `http://mech.vub.ac.be/teaching/info/mechatronica/finished_projects_2015/team_4/index.html`, 2017, [Online; accessed May 21, 2019].

[BK17] G. Bohouta and V. Këpuska, *Comparing Speech Recognition Systems (Microsoft API, Google API And CMU Sphinx)*, Int. Journal of Engineering Research and Application **2248-9622** (2017), 20–24.

[BYF13] D. P. Martin B. Yeh, A. Trakowski and J. Flohr, *CarlsenBot– Detailed Report (long post)*, `https://benpyeh.com/2013/05/02/carlsenbot-detailed-report-long-post/`, 2013, [Online; accessed May 21, 2019].

[CFVI19] Clker-Free-Vector-Images, *Chess Pieces Set Free Picture*, https://www.needpix.com/photo/20042/chess-pieces-set-symbols-game-pawn-queen-knight-bishop, 2019, [Online; accessed May 21, 2019].

[che] *Chessboard image*, https://cdn.shopify.com/s/files/1/1297/3303/products/standard-walnut-chess-board-21184254145_1024x1024.jpg?v=1516111679, [Online; accessed May 21, 2019].

[con19] *Elektromagnet Intertec ITS-PE2025-12VDC, 45 N, 12 V/DC, 6 W* , https://www.conrad.cz/elektromagnet-intertec-its-pe2025-12vdc-45-n-12-v-dc-6-w.k506161, 2019, [Online; accessed May 21, 2019].

[Dor19] S. Dorodnicov, *realsense2_camera package*, http://wiki.ros.org/realsense2_camera, 2019, [Online; accessed May 21, 2019].

[Fau18] J. Faust, *visualization_msgs/Marker*, http://docs.ros.org/api/visualization_msgs/html/msg/Marker.html, 2018, [Online; accessed May 21, 2019].

[Foo18] T. Foote, *geometry_msgs/Pose*, http://docs.ros.org/api/geometry_msgs/html/msg/Pose.html, 2018, [Online; accessed May 21, 2019].

[Fou19] S. Fourtané, *The Turk: Wolfgang von Kempelen's Fake Automaton Chess Player*, https://interestingengineering.com/the-turk-fake-automaton-chess-player, 2019, [Online; accessed May 21, 2019].

[GJ$^+$10] Gaël Guennebaud, Benoît Jacob, et al., *Eigen v3*, http://eigen.tuxfamily.org, 2010, [Online; accessed May 21, 2019].

[GJMSMCMC15] Sergio Garrido-Jurado, Rafael Muñoz-Salinas, Francisco Madrid-Cuevas, and Rafael Medina-Carnicer, *Generation of fiducial marker dictionaries using Mixed Integer Linear Programming*, Pattern Recognition **51** (2015).

[Its14] Itseez, *The OpenCV Reference Manual*, 2.4.9.0 ed., April 2014, [Online; accessed May 21, 2019].

[Its15] Itseez, *Open Source Computer Vision Library*, https://github.com/itseez/opencv, 2015, [Online; accessed May 21, 2019].

[kuk16] *LBR iiwa 7 R800, LBR iiwa 14 R820 Specification* , https://www.cobotware.ro/wp-content/uploads/2017/10/cobotware-specificatii-iiwa-kuka.pdf, 2016, [Online; accessed May 21, 2019].

[kuk19] *iiwa by KUKA*, https://store.clearpathrobotics.com/products/iiwa, 2019, [Online; accessed May 21, 2019].

[Lyo12] J. Lyons, *Mel Frequency Cepstral Coefficient (MFCC) tutorial*, http://www.practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/, 2012, [Online; accessed May 21, 2019].

[mar15] *Detection of ArUco markers*, https://docs.opencv.org/3.1.0/d5/dae/tutorial_aruco_detection.html, 2015, [Online; accessed May 21, 2019].

[Mey17] J. Meyer, *Chess Playing Robot*, http://www.raspberryturk.com/, 2017, [Online; accessed May 21, 2019].

[Mor00] B.S. Morse, *Lecture 4: Thresholding*, http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MORSE/threshold.pdf, 2000, [Online; accessed May 21, 2019].

[NUAD17] Y. S. Haruna N. U. Alka, A. A. Salihu and I. A. Dalyop, *VoiceControlled Pick and Place Robotic Arm Vehicle Using AndroidApplication*, American Journal of Engineering Research (AJER) **6** (2017), 207–215, [Online; accessed May 21, 2019].

[ope19] *Camera Calibration and 3D Reconstruction*, https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html, 2019, [Online; accessed May 21, 2019].

[pytch] *The Ultimate Guide To Speech Recognition With Python*, https://realpython.com/python-speech-recognition/#how-speech-recognition-works-an-overview, 2018, March, [Online; accessed May 21, 2019].

[Res17] Microsoft Research, *Automatic Speech Recognition - An Overview*, https://www.youtube.com/watch?v=q67z7PTGRi8, September 2017, [Online; accessed May 21, 2019].

[RRMSMC18] Francisco Romero Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer, *Speeded Up Detection of Squared Fiducial Markers*, Image and Vision Computing **76** (2018).

[Sho] K. Shoemake, *Quaternions*, http://www.cs.ucr.edu/~vbz/resources/quatut.pdf, [Online; accessed May 21, 2019].

[speer] *Generating Waveforms for Podcasts in Winds 2.0*, https://hackernoon.com/generating-waveforms-for-podcasts-in-winds-2-0-82a32a1c77fa, 2017, December, [Online; accessed May 21, 2019].

[Vel94] T. Veldhuizen, *Expression Templates*, https://web.archive.org/web/20050210090012/http://osl.iu.edu/~tveldhui/papers/Expression-Templates/exprtmpl.html, 1994, [Online; accessed May 21, 2019].

[VP18] L. Wagner V. Petrík, *capek_testbed package*, https://gitlab.ciirc.cvut.cz/capek/capek_testbed, 2018, [Online; accessed May 21, 2019].

[Č8] V. Číhala, *Využití silově poddajného robota na čistění bot*, Bachelor thesis, Czech Technical University in Prague, 2018.

[Še18] G. Šejnová, *language_ctrl package*, https://gitlab.ciirc.cvut.cz/tesarm11/imitrob_demo/tree/master/language_ctrl, 2018, [Online; accessed May 21, 2019].

[Šk17] R. Škoviera, *Introduction to Robot Operating System: with a little bit of Python: a.k.a. the most basic intro to ROS ever given*, December 2017.