Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Markus Bjønnes, Marius Nilsen

ROS 2 Integration of the ABB YuMi Dual-Arm Robot and the Zivid One 3D Camera for Autonomous Manipulation of Small Components

Master's thesis in Robotics and Automation

Supervisor: Lars Tingelstad

June 2020

NTNU
Norwegian University of Science and Technology

Preface

This thesis concludes five years of studies at the Department of Mechanical and Industrial Engineering at NTNU. We would like to express our gratitude towards our supervisor Lars Tingelstad for valuable advice and guidance the last two semesters. We would also like to thank Martin Ingvaldsen, at Zivid AS, for valuable input regarding the use of their cameras.

Work on this thesis was conducted during the Covid-19 pandemic. In accordance with NTNU policy, we did not have access to the laboratory facilities from 13th of March to the 4th of May.

[illegible] [illegible]

Sammendrag

Hovedmålet for denne avhandling er utviklingen av et robotsystem for ekstern styring av en toarmet ABB YuMi industrirobot. Robotsystemet skal utføre oppgaver som autonom håndtering av små komponenter i montasjeoperasjoner. Til dette formålet, presenterer vi metodikk og implementasjonsdetaljer for et ROS 2 basert system. Systemet integrer 3D datasyn for estimering av objekters posisjon og orientering, samt automatisk bevegelsesplanlegging.

Roboten styres ved hjelp av sine "Externally Guided Motion" og "Robot Web Services" grensesnitt. Gjennom disse styres roboten eksternt i sanntid fra et system implementert i ROS 2. For implementasjon av automatisk bevegelsesplanlegging brukes den initielle MoveIt 2 Betaen. Robotsystemet benytter en samplingsbasert og en lineær kartesisk baneplanlegginsmetode. Objekters posisjon og orientering innhentes fra nøyaktige punktskyer generert av en Zivid One 3D kamera. Forprossesering av punktskyer gjøres ved hjelp av programvarebiblioteket PCL, estimering av posisjon og orientering gjøres ved å benytte "point pair features". Styring av Zivid One 3D kameraet samt estimering av objekters posisjon og orientering er integrert i robotsystemet ved hjelp av ROS 2.

Systemets komponenter ble testet både isolert og samlet. Testing av objektdeteksjonsmetoden viste at denne pålitelig fant objekter. Videre viste testing at baneplanleggingsmetodene robust planla baner som unngikk hindringer i rommet. Systemvalidering ble gjennomført i et eksperiment hvor komponenter plukkes fra en boks og plasseres i en annen. Systemet klarte autonomt å gjennomføre oppgaven på en pålitelig måte, noe som indikerer at systemet kan benyttes i industrielle montasjeoperasjoner.

Abstract

The main objective of this thesis is the development of a system for external control of the ABB YuMi dual arm industrial robot. The robot system is intended to be applied for industrial assembly purposes, carrying out tasks such as the autonomous handling of small components. To this end, we present methodology and implementation details for a ROS 2 based system, integrating 3D computer vision for object pose estimation and automatic motion planning.

The robot is controlled using the Externally Guided Motion and Robot Web Services control interfaces. Through these, the robot is externally controlled in a real-time manner from a robot control system implemented in ROS 2. Automatic motion planning is implemented using the initial MoveIt 2 Beta in sampling-based and linear Cartesian motion planning pipelines. Object poses are obtained from accurate point clouds generated by a Zivid One 3D camera. Preprocessing of point clouds is implemented using the PCL software library and pose estimation is done using point pair features. Both control of the Zivid One 3D camera and pose estimation functionality is integrated into the robot system using ROS 2.

The implemented system components were tested both individually and in an integrated system configuration. Individual tests showed that the pose estimation pipeline was able to reliably find the pose of objects from point clouds. Furthermore, both motion planning pipelines were able to reliably plan motions, avoiding collisions with the environment. System integration was tested in a bin picking application. The system was able to autonomously and reliably perform the bin picking task, demonstrating the applicability of the system to industrial component handling tasks.

Contents

List of Figures

List of Tables

Acronyms

6DoF Six Degrees of Freedom.

API Application Programming Interface.

BVH Bounding Volume Hierarchies.

CAD Computer Aided Design.

CNN Convolutional Neural Network.

DDS Data Distribution Service.

EGM Externally Guided Motion.

FCL Flexible Collision Library.

GUI Graphical User Interface.

HTTP Hypertext Transfer Protocol.

ICP Iterative Closest Point.

IDE Integrated Development Environment.

KDL Kinematics and Dynamics Library.

LAN Local Area Network.

OMPL Open Motion Planning Library.

OS Operating System.

PCL Point Cloud Library.

QoS Quality of Service.

RANSAC Random Sample Consensus.

REST Representational State Transfer.

RGB Red Green Blue.

RGB-D Red Green Blue Depth.

ROS Robot Operating System.

ROS 2 Robot Operating System 2.

RRT Rapidly-Exploring Random Tree.

RWS Robot Web Services.

SDK Software Development Kit.

STL Stereolithography.

TCP Tool Center Point.

TCP/IP Transmission Control Protocol.

UDP User Datagram Protocol.

UML Unified Modelling Language.

URDF Universal Robot Description Format.

URI Uniform Resource Identifier.

Chapter 1

Introduction

In current manufacturing environments there exists a requirement to continuously increase the degree of automation in order to improve efficiency and reduce costs. Simultaneously, it is important to maintain the ability to rapidly adapt to product variations on the same production line, especially for small batch size production [21]. In particular, the assembly stage of a manufacturing process is both time consuming and in many cases the most labour intensive [20]. Assembly often involves the handling of small components, picked from bins, which require automation solutions with a higher level of autonomy than those traditionally employed in large scale productions industries where robots often carry out the same repetitive motions continuously.

To increase the degree of automation in difficult assembly and component handling tasks, robotics solutions such as the ABB YuMi has been developed. These collaborative robots allow for closer interaction between human and machine in an effort to increase the efficiency of both. Increasing the level of autonomy and degree of adaptability of these robotics solutions require integration of additional system capabilities such as decision making based on the surrounding environment.

To this end, integration of computer vision systems and motion planning algorithms capable of obstacle avoidance can be employed. This makes autonomous interaction with the robot's surroundings and safer operation in the presence of obstacles possible. Computer vision systems process digital imaging data giving machines visual perception, which can allow robots to gain awareness of it's environment. This awareness permits planning and execution of robot motions to be defined by simple task specifications.

In this thesis, we present a robot system capable of autonomous handling of small components. The system integrates a vision system utilizing a Zivid One 3D camera. The robot is externally controlled and a motion planning system capable of obstacle avoidance is implemented using the Moveit 2 Beta. System integration is done using ROS 2.

1.1 Objectives

The main objectives of this thesis are:

- Implement software which can be used to control the ABB YuMi from an external computer through it's external control interfaces.
- Implement functionality for automatic motion planning and obstacle avoidance.
- Implement software for integrating the Zivid One 3D camera into the robotic system.
- Implement software for integrating object pose estimation methods into the robotic system.
- Develop and implement a method for robotic hand-eye calibration using the Zivid One 3D camera.
- Implement a system for sensor-less estimation of external forces acting on the robot end-effector.
- Test and validate all system components both individually and in an integrated system configuration.

1.2 Contributions

The main contributions of this thesis are:

- An open source ROS 2 software solution for external control of the ABB YuMi.
- System integration of a Zivid 3D camera and pose estimation methods.
- Early adoption and testing of the Beta release for the MoveIt 2 Motion Planning Framework.
- Investigation and implementation of robotic hand-eye calibration using accurate point cloud data for extrinsic camera calibration.

1.3 Thesis structure

The thesis is structured as follows. Chapter 2 presents preliminary theory on robotics, computer vision and the ROS 2 framework. Chapter 3 investigates literature regarding methods for object recognition and pose estimation using computer vision, and discusses challenges for implementing these into a robotics system. A developed method using point clouds for robotic hand-eye calibration is also presented. In Chapter 4, the problem of motion and trajectory planning is characterized, and two motion planning pipelines using the MoveIt 2 Beta are outlined. Chapter 5 presents use cases for the measurement of external forces acting on the robot end-effector, and presents a method for estimation of these forces without the use of a dedicated sensor. Chapter 6 presents the implemented robotics system. This chapter is divided into two parts, one covering hardware and one covering the software implementations. Chapter 7 presents experiments carried out to evaluate the performance of the implemented system. Each of the main system components are tested individually in addition to a system integration test where a bin-picking task is carried out. The results of the experiments and overall system performance, are discussed in Chapter 8. This chapter also contains concluding remarks as well as suggestions for further work improving and using the implemented system.

Chapter 2

Preliminaries

The aim of this chapter is to provide the reader with the necessary theoretical knowledge regarding methods used for modelling of rigid bodies in 3D space, the fundamentals used in computer vision systems, and a conceptual description of ROS 2. The theoretical concepts presented in this chapter is gathered from several textbooks as well as papers published in widely accepted journals [60], [13], [39], [26], [27]. The majority of the contents of this chapter are based on the delivery in the specialisation project in the course TPK4560, at NTNU [9, 46].

2.1 Kinematic Modelling of Rigid Bodies

The kinematics of rigid bodies is a central concept used in modelling and control of robot manipulators. It provides a powerful tool for representing the position and orientation, or pose, of rigid bodies in three dimensional space. The term robot manipulator is used as a general term meaning a robotic arm. Robotic arms consist of rigid links connected together by joints. These joints can vary in type, but the most common ones are revoulute with one rotational degree of freedom and prismatic joints with one translational degree of freedom.

2.1.1 Pose of Rigid Bodies

Rigid bodies in 3D space have three positional degrees of freedom and three rotational degrees of freedom, adding up to a total of six degrees of freedom. Having a robust mathematical representation for the pose of rigid bodies is fundamental for the development of methods used in modelling kinematic chains. This representation is based on matrices in the special orthogonal group, $SO(3)$ and the special Euclidean group, $SE(3)$.

2.1.2 Rotation Matrices

Rotation matrices can be used to represent the difference in orientation of a coordinate system $\{a\}$ and a rotated coordinate system $\{b\}$. Coordinate frames in 3D space are represented as 3×3 matrices with each column being a unit vector. This set of column vectors make up an orthogonal basis for $\mathbb{R}^3$. The axes of the reference coordinate system can be represented as the columns of the identity matrix in $\mathbb{R}^{3\times3}$.

$$\{a\} = \begin{bmatrix} x_a & y_a & z_a \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

The rotated frame $\{b\}$ is obtained by applying the linear transformation R_{ab}, representing the rotation from frame $\{a\}$ to frame $\{b\}$. The elements of the rotation matrix R_{ab} is as shown below.

$$R_{ab} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{2.2}$$

Each column of the matrix representing coordinate frame $\{b\}$ are the the directional unit vectors of the the axes of frame $\{b\}$ given in the coordinates of frame $\{a\}$. $SO(3)$ has the following definition:

$$R \in SO(3) = \{R \in \mathbb{R}^{3\times3}, R^T R = I^{3\times3}, \det(R) = 1\} \tag{2.3}$$

From this it follows that

$$R^T = R^{-1} \tag{2.4}$$

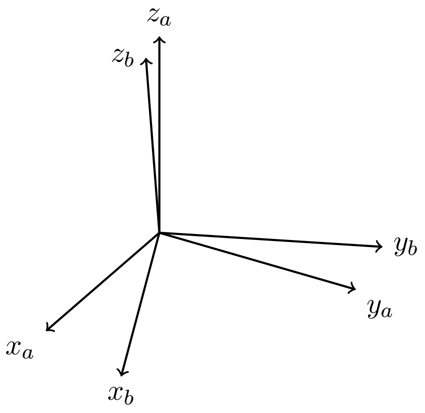


Figure 2.1: Coordinate frames $\{a\}$ and $\{b\}$.

2.1.3 Elementary Rotations

Elementary rotations are defined as rotations by an angle θ about one of the three principal axes of a coordinate frame. Elementary rotations about the z, y, and x axes are presented below:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.5}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \tag{2.6}$$

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}. \tag{2.7}$$

2.1.4 Rotations as Linear Transformations

Given the coordinate frames $a,\ b,\ c$ where the orientation of frame b in the coordinates of frame a is defined by the rotation matrix R_{ab}, and the orientation of frame c in the coordinates of frame b is defined by the rotation matrix R_{bc}, we can obtain the orientation of frame c in the coordinates of frame a as the rotation matrix R_{ac} composed from the intermediate rotations R_{ab} and R_{bc} as $R_{ac} = R_{ab}R_{bc}$. The interpretation of this operation is that frame a is first rotated to frame b by R_{ab}, frame b is then rotated by R_{bc} to frame c.

It is important to note that matrix multiplication operations for matrices in $SO(3)$ are associative: $(R_1R_2)R_3 = R_1(R_2R_3)$, but not commutative: $(R_1R_2 \neq R_2R_1)$. The interpretation of pre- or post multiplication by a rotation matrix is that pre-multiplcation results in a rotation about the current body frame, while a post-multiplication results in a rotation about a fixed reference frame.

2.1.5 Euler Angles

As there are three degrees of rotational freedom in 3D space, the parametrization of the orientation of a rigid body can be represented by three independent rotation angles. This parametrization is referred to as Euler angles. It is possible to construct several different representations of the orientation of a rigid body using Euler angles e.g. ZYZ, XYZ, XZX and ZYX, where X, Y and Z represent elementary rotations about the x, y and z axes respectively. ZYX-Euler angles are derived below.

Given a rigid body with orientation defined by the body frame b, initially aligned with the fixed space frame s. The orientation of frame b in the coordinates of frame a are parametrized by the triplet of angles (α, β, γ) in the following rotation operations.

- The rotation about the z-axis of the fixed frame s by the angle α, resulting in the body frame s',
- The rotation about the y-axis of the body frame s' by the angle β, resulting in the body frame s'',
- The rotation about the x-axis of the body frame s'' by the angle γ, resulting in the body frame b.

$$R_{sb} = R_z(\alpha)R_y(\beta)R_x(\gamma) =$$
$$\begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix} = \tag{2.8}$$
$$\begin{bmatrix} c_\alpha c_\beta & c_\alpha s_\beta s_\gamma - s_\alpha c_\gamma & c_\alpha s_\beta c_\gamma + s_\alpha s_\gamma \\ s_\alpha c_\beta & s_\alpha s_\beta s_\gamma + c_\alpha c_\gamma & s_\alpha s_\beta c_\gamma - c_\alpha s_\gamma \\ -s_\beta & c_\beta s_\gamma & c_\beta c_\gamma \end{bmatrix}$$

Here $c_\theta = \cos\theta, s_\theta = \sin\theta$ are used for the convenience of a more compact notation.

2.1.6 Skew Symmetric Representation of a Vector

A skew symmetric matrix $S \in \mathbb{R}^{n\times n}$ satisfies the condition that $S^T = -S$, meaning that the transpose of the matrix is equal to its negative. Given a vector $\boldsymbol{v} \in \mathbb{R}^3$, a skew symmetric matrix can be constructed from $\boldsymbol{v}$ as:

$$\boldsymbol{v}^\times = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix} \tag{2.9}$$

2.1.7 Rotations as Matrix Exponentials and Logarithms

The matrix exponential coordinates is a parametrization of a rotation matrix by a vector $\boldsymbol{k}\theta \in \mathbb{R}^3$ where $\boldsymbol{k}$ is the unit vector representing the axis of rotation in $\mathbb{R}^3$ and θ is a scalar value representing the angle of rotation about the axis $\boldsymbol{k}$. Writing θ and $\boldsymbol{k}$ separately gives the angle-axis representation of the rotation matrix. The matrix logarithm is represented by the skew symmetric matrix $\boldsymbol{k}^\times\theta$, and is a member of the Lie algebra, denoted $so(3)$, of the special orthogonal group,

$SO(3)$. The elements of the lie algebra $so(3)$ can be interpreted as the tangent space of the manifold $SO(3)$ at the identity element. The matrix logarithm is related to the rotation matrix by:

$$\boldsymbol{k}^{\times}\theta \in so(3) \rightarrow R \in SO(3) : e^{\boldsymbol{k}^{\times}\theta} = R \tag{2.10}$$

The matrix exponential, $e^{k^{\times}\theta}$, is calculated by Rodrigues' formula:

$$R(\boldsymbol{k}, \theta) = I + \sin(\theta)\boldsymbol{k}^{\times} + (1 - \cos(\theta))\boldsymbol{k}^{\times}\boldsymbol{k}^{\times} \tag{2.11}$$

where I is the identity matrix in $\mathbb{R}^{3\times3}$. Given a rotation represented by $R \in SO(3)$ the matrix exponential coordinate representation can be found by the following relation:

$$SO(3) \rightarrow so(3) : \; R \mapsto \log(R) = \boldsymbol{k}^{\times}\theta \tag{2.12}$$

The angle of rotation θ and the axis of rotation $\boldsymbol{k}$ are calculated from Equations (2.13) and (2.14).

$$\theta = \cos^{-1}\left(\frac{\text{trace}(R) - 1}{2}\right) \tag{2.13}$$

$$\boldsymbol{k} = \frac{1}{2\sin\theta}\begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \tag{2.14}$$

Here trace(R) is the sum of the elements on the principal diagonal of R and r_{ij} are the elements of R. Note that this holds only when trace$(R) \neq -1$ and trace$(R) \neq 3$, as it would result in $\sin\theta = 0$.

2.1.8 The Unit Quaternion

The unit quaternion provides a useful four-parameter representation for rotations or orientations in 3D space. The unit quaternion is defined as $Q(\eta, \boldsymbol{\epsilon})$ where:

$$\eta = \cos\left(\frac{\theta}{2}\right) \tag{2.15}$$

$$\boldsymbol{\epsilon} = \sin\left(\frac{\theta}{2}\right)\boldsymbol{k} \tag{2.16}$$

Here θ and $\boldsymbol{k}$ refer to the rotation angle and axis of rotation from Section 2.1.7 respectively. The scalar part η and the vector part ϵ make up the quaternion vector $Q = [\eta, \epsilon_x, \epsilon_y, \epsilon_z]$. For the unit quaternion, the following holds:

$$||Q|| = \sqrt{\eta^2 + \epsilon_x^2 + \epsilon_y^2 + \epsilon_z^2} = 1 \tag{2.17}$$

The quaternion product $Q_1 * Q_2$ is defined as:

$$Q_1 * Q_2 = [\eta_1\eta_2 - \boldsymbol{\epsilon_1}^T\boldsymbol{\epsilon_2}, \eta_1\boldsymbol{\epsilon_2} + \eta_2\boldsymbol{\epsilon_1}, \boldsymbol{\epsilon_1} \times \boldsymbol{\epsilon_2}] \tag{2.18}$$

Which corresponds to the product of the two rotation matrices R_1, R_2.

2.1.9 Homogeneous Transformations

Homogeneous transformation matrices in the special Euclidean group $SE(3)$ are the set of all 4×4 matrices on the form

$$T = \begin{bmatrix} R & \boldsymbol{t} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.19}$$

where $R \in SO(3)$ and $t \in \mathbb{R}^3$. Multiplication with homogeneous transformation matrices is associative: $(T_1T_2)T_3 = T_1(T_2T_3)$, but not commutative: $T_1T_2 \neq T_2T_1$. Homogeneous transformation matrices have three important uses.

- Represent the orientation and position of a rigid body.
- Change the reference frame in which a vector or frame is represented.
- Displace a vector or frame.

The inverse of a matrix $T \in SE(3)$ is also in $SE(3)$, this also holds for the product of two matrices in $SE(3)$. The inverse of a homogeneous transformation matrix can be calculated from:

$$T^{-1} = \begin{bmatrix} R & \boldsymbol{t} \\ 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} R^T & -R^T\boldsymbol{t} \\ 0 & 1 \end{bmatrix} \tag{2.20}$$

2.2 Kinematics of Open Chains

2.2.1 Forward Kinematics

Forward, or direct, kinematics are used to calculate the position and orientation of the robot end-effector based on the joint configuration. For a robot with n

variable joints the joint configuration vector $\boldsymbol{q}$ is written as:

$$\boldsymbol{q} = [q_1,\ q_2, ...,\ q_n] \tag{2.21}$$

where q_1 is the joint parameter of joint 1, q_2 is the joint parameter for joint 2 an so on. The forward kinematics of a kinematic chain can be expressed as the product of homogeneous transformation matrices, each depending on a single joint variable q_i.

$$T_{0n}(\boldsymbol{q}) = T_{01}(q_1)T_{12}(q_2)...T_{(n-1)n}(q_n) \tag{2.22}$$

Where $T_{(i-1)i}$ is the transformation from the coordinate frame attached to joint number $i-1$ to the coordinate frame attached to joint number i. The resulting matrix T_{0n} is the transformation from the coordinate frame attached to joint 1 to the coordinate frame attached to joint n.

The pose of the robot end-effector frame $\{t\}$ relative to the base frame $\{b\}$ is obtained from

$$T_{bt} = T_{b0}T_{0n}(\boldsymbol{q})T_{nt} \tag{2.23}$$

Where T_{b0} and T_{nt} are constant transformations. The transformation from the last robot joint to the end-effector frame T_{nt} has to be calibrated when using different end-effectors. There are several different methods used for deriving the forward kinematics of a robotic manipulator, such as the Denavit-Hartenberg (DH) convention [27] and the method using products of exponentials based on joint twists. The method of deriving the forward kinematics of an open chain manipulator based on the DH-convention based on is presented below.

The DH-convention uses four parameters $(a_i, d_i, \alpha_i, \theta_i)$ to define the pose of the frame attached to joint i in reference to the frame attached to joint $i-1$. The parameters are assigned when the manipulator is in a pre-defined zero-position, meaning that all the joint variables are set to zero ($\boldsymbol{q} = [0,\ 0, ...,\ 0]$). a_i and d_i are the respective translations along the x_{i-1} and z_{i-1} axes from the origin of frame $i-1$ to the origin of frame i. α_i is the rotation about the x_{i-1} axis and θ_i is the rotation about the z_{i-1} axis. For revolute joints the variable $q_i = \theta_i$ and for prismatic joints the joint variable is $q_i = d_i$. The joint transformation between joint $i-1$ and i is a function of the joint variable q_i and is calculated as

$$T_{(i-1)i}(q_i) = \begin{bmatrix} \cos\theta_i & -\sin\theta_i cos\alpha_i & \sin\theta_i \sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.24}$$

2.2.2 Inverse Kinematics

Inverse kinematics is the process of solving the joint configuration vector $\boldsymbol{q}$ of the manipulator, given a desired pose in the task space. The problem of solving the inverse kinematics for a manipulator can be formulated as finding a solution $\boldsymbol{q}$ which satisfies the forward kinematics of the manipulator $f(\boldsymbol{q}) = X$, for a given pose $X \in SE(3)$.

2.2.3 Analytical Inverse Kinematics

Analytical inverse kinematics seeks calculate the manipulator joint variables given the desired pose of the end-effector frame in the workspace. Analytic solutions generally use the geometric relationships between manipulator links and joints to find a general expression for each joint variable. A brief example showing these expressions for a simple planar arm consisting of two links is provided to illustrate the concept.

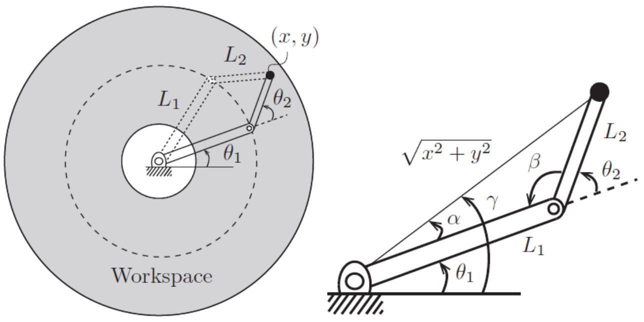


Figure 2.2: Two link planar arm. Figure from [39].

Considering the manipulator structure in Figure 2.2 with the joint variables θ_1, θ_2 link lengths $L1, L2$ and end-effector location x, y. The orientation of the end-effector is not considered. From the law of cosines we get.

$$L_1^2 + L_2^2 - 2L_1L_2\cos(\beta) = x^2 + y^2 \tag{2.25}$$

re-arranging this we find the expressions for α and β

$$\beta = \cos^{-1}\left(\frac{L_1^2 + L_2^2 - x^2 - y^2}{2L_1L_2}\right) \tag{2.26}$$

$$\alpha = \cos^{-1}\left(\frac{L_1^2 - L_2^2 + x^2 + y^2}{2L_1\sqrt{x^2+y^2}}\right) \tag{2.27}$$

The angle γ is found by

$$\gamma = \arctan\left(\frac{y}{x}\right) \tag{2.28}$$

taking care to consider which quadrant the end-effector is in, in order to get the correct angle. The two possible solutions shown in 2.2 consists of a configuration where the arm is to the left of the end-effector position, and a configuration where the arm is to the right of the end-effector position.

$$\text{Right: } \theta_1 = \gamma - \alpha,\ \theta_2 = \pi - \beta \tag{2.29}$$

$$\text{Left: } \theta_1 = \gamma + \alpha,\ \theta_2 = \beta - \pi \tag{2.30}$$

2.2.4 Numerical Inverse Kinematics

The numerical inverse kinematics problem is a consists of finding the roots of a non-linear matrix equation. There exist several iterative methods for solving such an equations. One method is the Newton-Raphson method which uses the first order Taylor expansion for a differentiable function $g(\boldsymbol{q})$ to solve the inverse problem. The Newton-Raphson method is briefly presented in the following.

Consider the desired, known, end-effector pose x_d, associated with an unknown joint-configuration $\boldsymbol{q_d}$, and the current end-effector pose x_i, associated with the current joint-configuration $\boldsymbol{q_i}$ such that $x_i = f(\boldsymbol{q_i})$. Where f is the forward kinematics function for the manipulator. The problem to be solved by the Newton-Raphson algorithm is then formulated as

$$g(\boldsymbol{q}) = x_d - f(\boldsymbol{q}) = 0 \tag{2.31}$$

where the solution $\boldsymbol{q} = \boldsymbol{q_d}$ is desired. The first order Taylor series expansion for the kinematics, given an initial guess $\boldsymbol{q_i}$ is

$$x_d = f(\boldsymbol{q_d}) = f(\boldsymbol{q_i}) + \left.\frac{\partial f}{\partial \boldsymbol{q}}\right|_{\boldsymbol{q_i}} (\boldsymbol{q_d} - \boldsymbol{q_0}) = f(\boldsymbol{q_0}) + J(\boldsymbol{q_i})\Delta\boldsymbol{q_i} \tag{2.32}$$

where $J(\boldsymbol{q_i})$ is the manipulator Jacobian defining the linear transformation $\mathcal{V} = J(\boldsymbol{q})\dot{\boldsymbol{q}}$ from the joint velocities to the end-effector twist, evaluated at $\boldsymbol{q_i}$. Equation (2.32) is manipulated to solve for $\Delta\boldsymbol{q_i}$

$$\Delta\boldsymbol{q_i} = J^{\dagger}(\boldsymbol{q_i})(x_d - f(\boldsymbol{q_i})) \tag{2.33}$$

where $J^{\dagger}(\boldsymbol{q_i})$ is the pseudo inverse of the Jacobian matrix to account for manipulators where the number of joints, $n \neq 6$, resulting in a non-square Jacobian. $\Delta\boldsymbol{q_i}$ is used as the update term for the iterative solver so that $\boldsymbol{q_{i+1}} = \boldsymbol{q_i} + \Delta\boldsymbol{q_i}$. The joint vector is updated in this manner until Equation (2.31) is satisfied within a predefined error threshold ε so that $g(\boldsymbol{q}) < \varepsilon$, resulting in the convergence $\boldsymbol{q} \rightarrow \boldsymbol{q_d}$.

2.3 Dynamic Modelling of Open Chains

Modelling of a manipulator's dynamics are used to account for the forces and torques which result in the motion described by the kinematics. This dynamic model is used in the design of the actuators, and to derive control schemes for the manipulator.

The general dynamics equation for an open chain manipulator consisting of n joints is

$$\boldsymbol{\tau} = M(\boldsymbol{q})\ddot{\boldsymbol{q}} + C(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}} + g(\boldsymbol{q}) \tag{2.34}$$

Where $\boldsymbol{\tau} \in R^n$ is the actuator torques, $M(\boldsymbol{q})\ddot{\boldsymbol{q}}$ is the torque contribution from joint-acceleration, $C(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}}$ is the torque contribution from Coriolis forces and $g(\boldsymbol{q})$ is the torque contribution from gravity.

As with forward and inverse kinematics, the concepts of forward and inverse dynamics are useful. Forward dynamics is the process of calculating the manipulator's joint-acceleration given the joint position and velocity $(\boldsymbol{q}, \dot{\boldsymbol{q}})$, in a given joint-state, and the applied joint-torques. This is described by the following equation.

$$\ddot{\boldsymbol{q}} = M^{-1}(\boldsymbol{q})(\boldsymbol{\tau} - C(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}} - g(\boldsymbol{q})) \tag{2.35}$$

The inverse dynamics problem is the process of calculating the joint-torques required to achieve a desired joint-acceleration given the current joint-state $(\boldsymbol{q}, \dot{\boldsymbol{q}})$. The inverse problem is described by Equation (2.34).

2.4 Computer Vision

Making use of cameras in robotics systems enables the robot to interact autonomously with its environment. Some of the central methods and models in computer vision are presented in this section.

2.4.1 Pinhole Camera Model

The pinhole camera model shown in Figure 2.3 is commonly used for modelling a real camera for use in computer vision applications. It models a simple camera where light is emitted through a pin-hole in the camera center, this simplification means that each point in the scene is projected as a single point in the image plane. Given a camera coordinate frame denoted $\{c\}$ and a world coordinate frame denoted by $\{w\}$, as shown in Figure 2.4. A point p in the scene with homogeneous coordinates given in the the camera frame is defined as

$$\tilde{\boldsymbol{r}}_{cp}^{c} = \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} \tag{2.36}$$

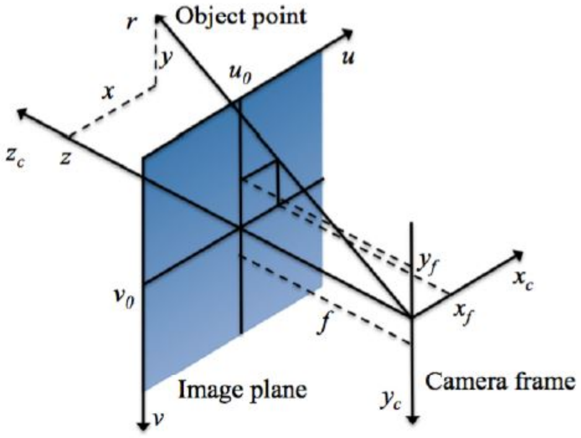


Figure 2.3: Pinhole camera model.

and the position of the same point in the world coordinate frame is

$$\tilde{\boldsymbol{r}}_{wp}^{w} = \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} \tag{2.37}$$

the transformation of reference frames for this point is given by

$$\tilde{\boldsymbol{r}}_{cp}^{c} = T_{cw}\tilde{\boldsymbol{r}}_{wp}^{w} \tag{2.38}$$

where T_{cw} is the homogeneous transformation matrix representing the transformation from $\{c\}$ to $\{w\}$.

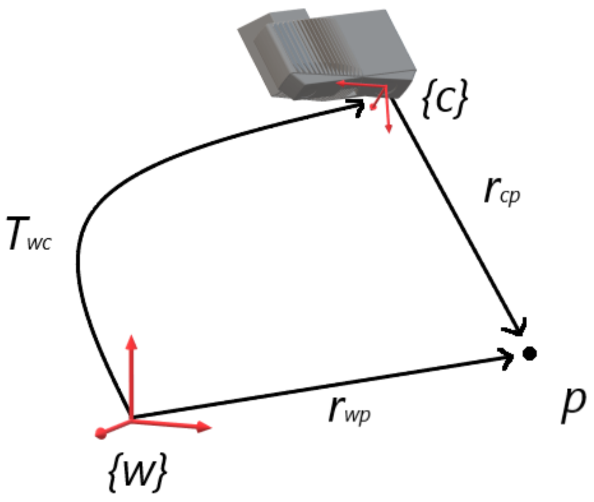


Figure 2.4: Camera and world coordinate systems.

The normalized coordinates of the point represented by $\boldsymbol{r}_{cp}^{c}$ projected onto the image plane is given by

$$\tilde{\boldsymbol{s}} = \begin{bmatrix} s_x \\ s_y \\ 1 \end{bmatrix} = \frac{1}{z_c}\boldsymbol{r}_{cp}^{c} = \frac{1}{z_c}\begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = \begin{bmatrix} \frac{x_c}{z_c} \\ \frac{y_c}{z_c} \\ 1 \end{bmatrix} \tag{2.39}$$

The conversion between homogeneous and non-homogeneous vector coordinates can be done by

$$\boldsymbol{r} = \Pi\tilde{\boldsymbol{r}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} r_x \\ r_y \\ r_z \\ 1 \end{bmatrix} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} \tag{2.40}$$

The normalized image coordinates can now be calculated as

$$\tilde{\boldsymbol{s}} = \frac{1}{z_c}\Pi\tilde{\boldsymbol{r}}_{cp}^{c} = \frac{1}{z_c}\Pi T_{cw}\tilde{\boldsymbol{r}}_{wp}^{w} \tag{2.41}$$

The corresponding pixel coordinate values are calculated from the following.

$$u = \frac{f}{\rho_w}s_x + u_0 \tag{2.42}$$

$$v = \frac{f}{\rho_h} s_x + v_0 \tag{2.43}$$

Here f is the focal length of the camera, ρ_w and ρ_h is the width and height of each pixel, and u_0 and v_0 is the image plane center coordinates. Conversion from normalized image coordinates to homogeneous pixel coordinates is possible using

$$\tilde{\boldsymbol{p}} = K\tilde{\boldsymbol{s}} \tag{2.44}$$

where K is the camera intrinsic parameter matrix.

$$K = \begin{bmatrix} \frac{f}{\rho_w} & 0 & u_0 \\ 0 & \frac{f}{\rho_h} & v_0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.45}$$

Equations (2.41) and (2.44) can be combined to the projective camera transformation.

$$z_c\tilde{\boldsymbol{p}} = K\Pi T_{cw}\tilde{\boldsymbol{r}}^w_{wp} \tag{2.46}$$

Which enables the calculation of pixel coordinates corresponding to a point $\tilde{\boldsymbol{r}}^w_{wp}$, given the relative transformation between $\{c\}$ and $\{w\}$. This formulation is the basis for solving of the inverse problem where the pixel values of a point is known, and the position in the coordinates of the fixed world frame $\{w\}$ is desired.

2.4.2 Corner Detection

In computer vision applications, the ability to detect distinct points on an object reliably under varying lighting conditions and from different viewing angles is very important for several applications including calibration, tracking and pose estimation. A corner can be formalized as the point at which two edges meet. Therefore an important step in any corner detection algorithm requires the detection of edges in an image.

Edges are usually found by finding the boundaries where the gradient of pixel intensity and color is abrupt. One way of doing this is by convolving the image with a gradient finding patch for x, and y directions. For instance the Sobel edge detector uses the patches in Equation (2.47) and Equation (2.48) to find the points at which the image gradient is large in the x and y directions respectively.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.47}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{2.48}$$

The edges found by this method can be used to efficiently find corners in an image using a variety of different corner detection algorithms like Harris [26], CSS [41] or SUSAN [61].

2.4.3 Camera Calibration

Intrinsic camera calibration is the process of establishing the camera's intrinsic parameters. The model used in Section 2.4.1 is simplified in the sense that the light passes through a pin-hole directly to the sensor. In a real camera, lenses focus the light in a way not accounted for in the pin-hole model, this needs to be corrected for by finding a set of distortion coefficients. The calibration parameters obtained by intrinsic calibration are the ones in Equation (2.45) accounting for focal distance, principal distance and the coordinates of the principal point. The distortion coefficients account for radial and tangential distortion, these are used to remove distortion effects from the image.

Extrinsic camera calibration solves the problem of estimating the pose of the camera in relation to the calibration object. This is a required step in the two step camera calibration algorithm presented in [64]. This algorithm provides efficient camera calibration based on co-planar calibration points. The extrinsic camera calibration is also central in solving the hand-eye calibration problem.

2.4.4 Point Clouds

A point cloud can be defined as a set consisting of n points having positions in 3D space $\boldsymbol{p_i} = [x_i, y_i, z_i]^T$. For generation of point clouds the accurate estimation of the distance of a point from the camera along the optical axis, z_i, is critical as this allows for calculation of the 3D position of the point relative to the camera using Equation (2.46). This distance can be estimated using a variety of methods such as stereo vision, time-of-flight range finders or structured light 3D scanners. Point clouds generated using these methods will often have an associated image color intensity for each point, $(I_{rgb})_i = [r_i, g_i, b_i]^T$, resulting in the notation $[(I_{rgb})_i,\ \ \boldsymbol{p_i}]$ for each point in the image.

2.4.5 Mathematical Description of Planes

A plane $\boldsymbol{\pi} = [a, b, c, d]^T$, with normal vector $\boldsymbol{n} = [a, b, c]^T$ at a distance $\frac{d}{||\boldsymbol{n}||}$ from the origin, is defined as the set of all points $\boldsymbol{p} = [x, y, z]^T$ satisfying

$$ax + by + cz + d = 0 \tag{2.49}$$

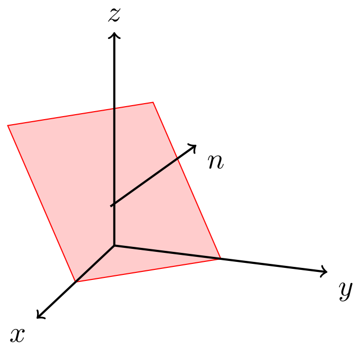


Figure 2.5: Plane with normal vector, $\boldsymbol{n}$.

2.4.6 Fitting a Plane to a Set of Points

If a homogeneous point $\tilde{\boldsymbol{p}}_i = [x_i, y_i, z_i, 1]^T$ is on the plane $\boldsymbol{\pi}$, the following holds.

$$\tilde{\boldsymbol{p}}_{\boldsymbol{i}}^{\,T}\boldsymbol{\pi} = 0 \tag{2.50}$$

This property can be used to find the best fit plane corresponding to a set of n homogeneous points. Following the derivation in [19], define a matrix $A = [p_1,\ p_2, ...,\ p_n]^T \in \mathbb{R}^{n\times 4}$. The plane π must satisfy

$$A\boldsymbol{\pi} = \begin{bmatrix} \boldsymbol{p_1}^T \\ \boldsymbol{p_2}^T \\ \vdots \\ \boldsymbol{p_n}^T \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = 0 \tag{2.51}$$

The singular value decomposition of A is

$$A = \sigma_1 \boldsymbol{u_1}\boldsymbol{v_1}^T + \sigma_2 \boldsymbol{u_2}\boldsymbol{v_2}^T + \sigma_3 \boldsymbol{u_3}\boldsymbol{v_3}^T + \sigma_4 \boldsymbol{u_4}\boldsymbol{v_4}^T \tag{2.52}$$

where $\sigma_1 > \sigma_2 > \sigma_3$ and $\sigma_4 = 0$. The only non-trivial solution for the plane $\boldsymbol{\pi}$ is along $\boldsymbol{v_4}$ giving the solution

$$\boldsymbol{\pi} = k\boldsymbol{v_4} \tag{2.53}$$

for some scale factor k. If the points in A are all on the plane, the solution is exact, otherwise it is the best fit solution minimizing the absolute orthogonal distance from each point to the plane.

2.4.7 Finding the Transformation Between two Point Clouds

Given two point clouds, $A \in \mathbb{R}^{n\times 4}$ and $B \in \mathbb{R}^{n\times 4}$, with n corresponding homogeneous points denoted $\boldsymbol{a_i}$ and $\boldsymbol{b_i}$ respectively the optimal transformation between them can be found by the minimization problem in Equation (2.57).

The correspondence between the point clouds is formulated as

$$B = TA \tag{2.54}$$

$$T = \begin{bmatrix} R & \boldsymbol{t} \\ 0 & 1 \end{bmatrix} \in SE(3) \tag{2.55}$$

Which can be written as

$$\boldsymbol{b_i} = Ra_i + \boldsymbol{t} \tag{2.56}$$

The expression to be minimized is the mean sum of square errors between the points in B and the transformed points TA.

$$\eta = \frac{1}{n}\sum_{i=1}^{n} \|T\boldsymbol{a_i} - \boldsymbol{b_i}\|^2 = \frac{1}{n}\sum_{i=1}^{n} \|R\boldsymbol{a_i} + \boldsymbol{t} - \boldsymbol{b_i}\|^2 \tag{2.57}$$

The optimal rotation is found by centring the two point clouds in the origin by subtracting the centroid (Equation (2.58)) of the point cloud from each point, then finding the best fit rotation by using the solution to orthogonal Procrustes problem (Equation (2.62)), which maximizes Equation (2.59).

$$\boldsymbol{c_A} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{a_i}, \quad \boldsymbol{c_B} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{b_i} \tag{2.58}$$

$$\max_{R} \operatorname{trace}(RH) \tag{2.59}$$

$$H = (A - \boldsymbol{c_A})(B - \boldsymbol{c_B})^T \tag{2.60}$$

The optimal rotation, R, is then found from the singular value decomposition of H:

$$U\Sigma V^T = \operatorname{svd}(H) \tag{2.61}$$

$$R = VSU^T \tag{2.62}$$

where S is the Umeyama correction (Equation (2.64)) to ensure that R is on

$SO(3)$. The optimal translation is then found from:

$$\boldsymbol{t} = -R\boldsymbol{c_A} + \boldsymbol{c_B} \tag{2.63}$$

$$S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \det(VU^T) \end{bmatrix} \tag{2.64}$$

It should be noted that this translation is dependent on the solution for $R \in SO(3)$, which means that the resulting translation is dependent on the optimization of the rotational problem in Equation (2.59).

2.4.8 Iterative Closest Point

The Iterative Closest Point (ICP) algorithm [8] uses an iterative approach to match two point clouds. The goal of is to find the transformation between the captured point cloud A and the model point cloud B, which minimizes the summed square distance of each matched point. The algorithm is comprised of the following steps. For each point $\boldsymbol{b_i}$ in B find the point y in A which minimizes the euclidean distance between the two points so that

$$d(b_i, y) = \min_{y \in A} ||\boldsymbol{b_i} - y|| \tag{2.65}$$

Compute the transformation between A and B using the matches and apply this transformation to each point in A. Iterate until the mean square error of matched points is below a desired threshold.

2.4.9 Structured Light 3D Scanners

Structured light 3D scanners is a category of depth sensors for imaging purposes which utilizes triangulation of a known light pattern to determine depth. The scanner consists of a light projector and a offset camera with known relative position and orientations to one another. The light is projected onto the scene and the deformation in the light pattern is used to triangulate the distance to the object on which the light shines.

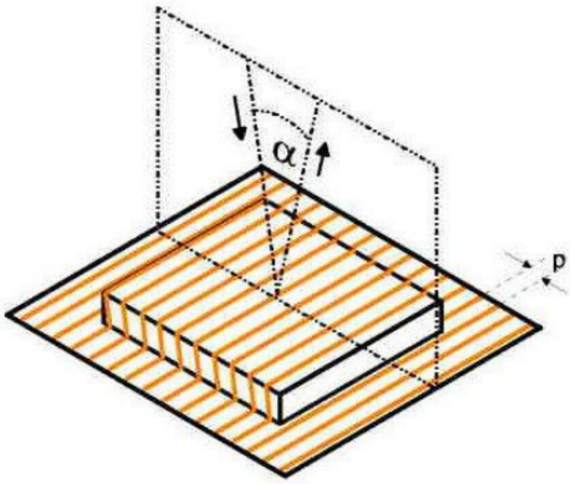


Figure 2.6: Illustration of triangulation principle for a structured light scanner, figure from [24].

Using the principle illustrated in Figure 2.6 the distance z from the camera to the object is calculated from the angle α and the displacement of projected lines p as

$$\boldsymbol{z} = \frac{\boldsymbol{p}}{\tan \alpha} \tag{2.66}$$

2.5 Robot Operating System 2

Robot Operating System 2 (ROS 2) is the second version of the Robot Operating System (ROS). The initial distribution of ROS was developed by Willow Garage in 2007 with the goal to reduce the need for re-writing code in robotics research and have since been widely adopted. ROS is not an operational system is the traditional sense, but functions as a flexible framework for writing robotics software. It provides a build system, a collection of software packages, tools and a set of conventions for code organization [4]. The initial distribution of ROS 2 was released in 2018. The distribution used in this thesis, Eloquent Elusor, was released in 2019.

ROS 2 utilize a file system whose main components are packages, metapackages, and workspaces. Packages are the most fundamental unit for organizing software, and is the smallest self-contained unit in the file system. A package might contain source code, configuration files or external libraries. The goal of the package is to organize software in a reusable and easy-to-consume manner. Related packages can be grouped and represented by metapackages. Metapackages do not install any files and are used to simply aggregate packages into groups. The directory containing a project's packages and metapackages is called a workspace. When using the ROS 2 build system, the build commands are called for a workspace, building the packages located within the root workspace folder. Built code can be utilized across workspaces.

A system built using ROS 2 consists of a number of processes, potentially on a number of different hosts, connected at runtime in a peer-to-peer topology. ROS 2 applications are built using nodes, components and libraries. A node is a process that performs computation, typically representing a subprogram within the ROS 2 system. In a topology context, node is interchangeable with 'software module'. A component is a container process which may contain several executables and nodes. Composition of multiple nodes into a component is typically used to reduce overhead on the communication between the nodes. Some code entities are not executable code, but defines useful resources, these are defined as libraries.

Nodes contained within different processes can communicate via topics, services and actions. Communication over topics are message oriented in the publish-subscribe pattern. A topic is a named data stream to which nodes can publish messages of a specified type. Topics support many-to-one, one-to-many, one-to-one and many-to-many communication. Communication is anonymous, meaning publishers publish to a topic without knowledge of the subscribers, and vice versa. Topics are typically used for continuous data streams like sensor data or robot state. Services are based on the client-server paradigm, where one node advertise a service, allowing other nodes to make us of the offered service. Services are named and typed, one-to-one and are typically used for remote procedure calls. Services in ROS 2 are asynchronous, meaning a service call is non-blocking. Actions are a combination of topic and service communication. An action is initiated in the same manner as a service, however the server can, upon request from the client, stream feedback and status over topics to the client. Additionally the client have the opportunity to cancel the requested goal. Actions are asynchronous and typically used for relatively time consuming tasks, like moving the robot towards a goal.

Lifecycle nodes are a special type of nodes allowing for runtime node management. Lifecycle nodes share the same base interfaces of regular ROS 2 nodes. Managing the lifecycle of nodes consist of controlling it's transition through a set of states. The states are UNCONFIGURED, INACTIVE, ACTIVATE, FINALIZED. Transitions are invoked using ROS 2 services provided by the lifecycle node interface. The states and transitions of the lifycle is depicted in fig. 2.7.

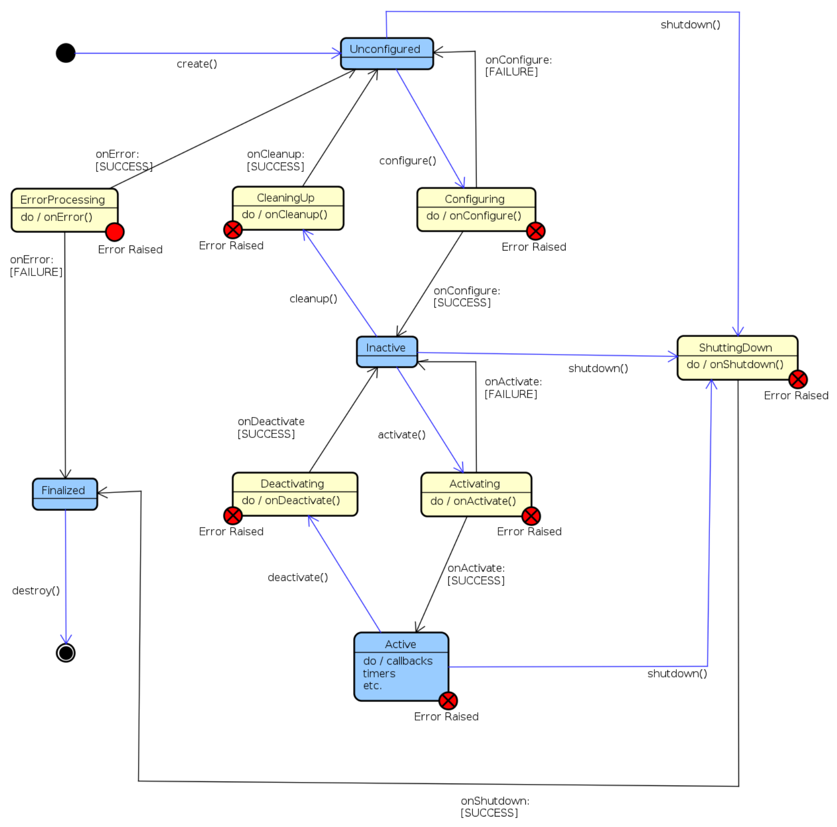


Figure 2.7: State and transition diagram for nodes with managed life cycles. Figure from [40].

ROS 2 uses a Data Distribution Service (DDS) as middleware. DDS is a data-centric end-to-end middleware standard used for dependable low latency data exchange [69]. DDS employs a virtual data space which, to the applications, appear as native memory. Applications read and write in this data space via an application programming interface as if it was in local storage. In reality, the DDS sends messages to update the appropriate store on the remote nodes, effectively ensuring operations are performed on the data directly instead of copies or reconstructions. Moreover, a DDS middleware is inherently decentralized, provides dynamic discovery of participants and supports specification of Quality of Service (QoS) for communication channels. Meaning, in ROS 2, a fully decentralized system can be realized. Furthermore, topics can be configured with QoS settings, meaning topics can be configured after prioritization, or to enhance a specific property, e.g. reliability or speed. This, in combination with copy-less data transfer, render ROS 2 more suited for real-time applications.

Chapter 3

Pose Estimation

The accurate estimation of Six Degrees of Freedom (6DoF) object poses in the robot workspace is an important part of bin-picking applications. Pose estimation methods making use of intensity images, point cloud data, or both in combination, is an active field of research with a magnitude of different approaches. This chapter is divided into two parts. The first part presents an overview of available methods for object pose estimation, this serves as a foundation for discussion regarding implementing pose estimation in an industrial bin-picking system. The second part is about hand-eye calibration, which is critical in integrating the pose estimation system with the overall robot system. Here, a method developed for hand-eye calibration of depth sensors, making use of accurate point cloud data is presented.

It is useful to categorize different methods for object pose estimation based on the base method they employ. These categories are geometry based, template based and deep learning based. As regards reviews of current state-of-the-art methods Hodaň et al. [31] conducted a comprehensive study comparing various different approaches for pose estimation. Their findings show that geometry based methods using point pair features [17, 16, 67] generally outperform other approaches under most conditions. It should however be noted that this conclusion reflects the state-of-the-art before 2018. More resent developments based on deep learning [70, 68] show promising results when compared with geometry and template based methods.

Comparisons between various pose estimation methods are based the on accuracy of the estimated pose evaluated against a known ground truth pose. In [30] an evaluation metric based on Visible Surface Discrepancy is proposed, calculating the estimation error over the visible surface of the object. If the ground truth pose for an object is known the accuracy may be measured as the discrepancy between the predicted pose and the ground truth. Another important evaluation metric is estimation time, which directly affects the rate at which the system can process imaging data. This may be an important consideration to avoid having bottlenecks in a robotics system.

3.1 Geometry Based Approaches

Geometry based methods aim to match geometric features sampled from 3D-model representations of an object, with a point cloud captured from a 3D imaging system.

3.1.1 Point Pair Feature Matching

Object pose estimation based on global model description using oriented point pair features was introduced by Drost et al. [17]. This model is matched with a scene using a fast voting scheme, where similar points pair features in the model and the scene are grouped together.

Creation of, and matching using point pair features assumes that both the model and the input scene consist of a finite number of 3D points with an associated surface normal. A point pair feature generated by two points $\boldsymbol{m_1}$ and $\boldsymbol{m_2}$ is defined using the vector $\boldsymbol{d}$ between the two points, and the two surface normals $\boldsymbol{n_1}$ and $\boldsymbol{n_2}$ as

$$\boldsymbol{F}(\boldsymbol{m_1}, \boldsymbol{m_2}) = (||\boldsymbol{d}||, \angle(\boldsymbol{n_1}, \boldsymbol{d}), \angle(\boldsymbol{n_2}, \boldsymbol{d}), \angle(\boldsymbol{n_1}, \boldsymbol{n_2})) = (F_1, F_2, F_3, F_4) \tag{3.1}$$

Where F_1, F_2, F_3, F_4 denote the four parameters in reference to Figure 3.1, and $\angle(\boldsymbol{a}, \boldsymbol{b}) \in [0, \pi]$.

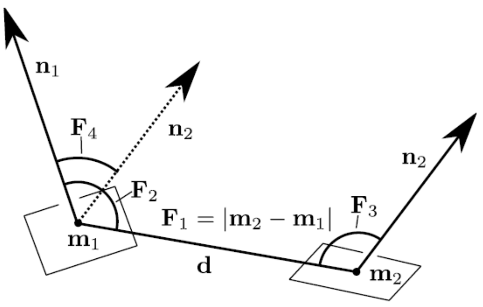


Figure 3.1: Illustration of how point pair features are generated from a pair of points. Figure from [17]

A global model descriptor for the object is built by generating point pair features for all point pairs on the model surface. The points used to represent the surface of the object is sampled from a 3D CAD model. Similar point pair features are then mapped to the same sets. This means that if the point pair features $\boldsymbol{F_1}(\boldsymbol{m_1}, \boldsymbol{m_2}), \boldsymbol{F_2}(\boldsymbol{m_3}, \boldsymbol{m_4}), \boldsymbol{F_3}(\boldsymbol{m_5}, \boldsymbol{m_6})$ are similar they well be stored in the same set $S_i = \{\boldsymbol{F_1}, \boldsymbol{F_2}, \boldsymbol{F_3}\}$. All the sets representing the model descriptor are stored in a hash table, which allows for constant time searches using a similar point pair feature in the test scene as the key.

At test time an efficient voting scheme is used to match the model descriptor to the scene. This method seeks to maximize the number of points in the scene which lie on the model by finding the best fit local coordinates. For this purpose a fixed reference point $\boldsymbol{s_r}$ is used. This point is paired with every other point in the scene $\boldsymbol{s_i} \in S$, the trained model descriptor is the searched for point pairs which match the scene. For each match $(\boldsymbol{m_r}, \boldsymbol{m_i})$ on the model, the transformation which maps to the corresponding points $(\boldsymbol{s_r}, \boldsymbol{s_i})$ in the scene is calculated. This voting scheme requires the reference points $\boldsymbol{s_i}$ to lie on the surface of the model to be found. This necessitates the use of multiple reference point in order to increase the likelihood of at least one of them being part of the model surface.

Practical implementations of pose estimation using point pair features requires the tuning of several parameters in the training stage, and in the matching stage. The goal is to determine a set of parameters which optimise the trade-off between speed and accuracy at test time. These parameters are, relative sampling step: the density of uniformly sampled surface points during model training relative to the model diameter. Relative scene sampling step: the portion of points in the test scene to be used for matching, increasing this value increases the robustness of matching at a cost of matching time. Relative distance step: determines how finely the hash table buckets are discretized, using a to coarse discretization may lead to matching of non-similar point pairs and using a to fine discretization can reduce the matching rate to much. Additionally parameters related to point cloud normal computation and parameters for ICP pose refinement (Section 2.4.8) may have an impact on both accuracy and speed.

3.1.2 Multimodal Point Pair Features

In order to increase the robustness of the pose estimation, the technique presented in [17] was extended using a multimodal feature descriptor for object surface and silhouettes [16]. This descriptor uses edges detected in an intensity image to find geometric 3D edges in the captured scene. Multimodal point pair features for a point on the geometric edge $\boldsymbol{e}$ and a reference point on the object surface $\boldsymbol{r}$ in the perspective-corrected intensity image is represented by the four parameter feature vector

$$F(\boldsymbol{e},\boldsymbol{r}) = (d(\boldsymbol{e},\boldsymbol{r}), \alpha_d, \alpha_n, \alpha_v) \tag{3.2}$$

where $d(\boldsymbol{e},\boldsymbol{r})$ is the metric distance between the points, $\alpha_d = \angle(\boldsymbol{e_d}, \boldsymbol{e} - \boldsymbol{r})$ is the angle between the edge gradient at point $\boldsymbol{e}$ and the vector from $\boldsymbol{e}$ to $\boldsymbol{r}$, $\alpha_n = \angle(\boldsymbol{n_r}, \boldsymbol{e} - \boldsymbol{r})$ is the angle between the surface normal vector at $\boldsymbol{r}$ and the vector from $\boldsymbol{e}$ to $\boldsymbol{r}$, $\alpha_v = \angle(\boldsymbol{n_r}, \boldsymbol{v_r})$ is the angle between the surface normal at $\boldsymbol{r}$ and the directional vector towards the camera at $\boldsymbol{r}$. The feature vector components are depicted in Figure 3.2.

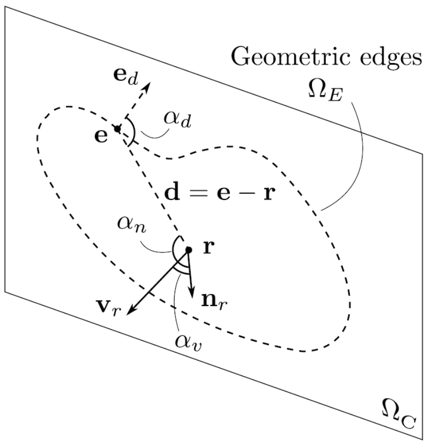


Figure 3.2: Components of a multimodal point pair feature. Figure from [16].

Model training consists of generating multimodal point pair features from multiple viewpoints sampled uniformly from a sphere around the object. The features are generated by rendering the model from different viewpoints and detecting geometric edges. Multimodal point pair features are then calculated for each edge point, and each reference point on the visible surface. The features are then stored in a hash table similar to [17].

3.2 Template based matching

Object pose estimation methods based on templates aim to match pre-generated object templates in an image or point cloud. Templates can be created using several different image properties, such as edges, image intensity, image gradients or surface normals [32]. Template matching methods for pose estimation in cluttered scenes have been shown to yield good results by [31] when compared to both learning based approaches and approaches based on point pair features.

3.2.1 LINEMOD

LINEMOD or multimodal-LINE is a method presented in [28] utilizing image features in different modalities to create templates for object matching. Their particular method uses image gradients extracted from a three channel RGB intensity image, and surface normals calculated from a point cloud generated from a 3D scanner, as illustrated in Figure 3.3.

Templates are generated from a set of reference images $\{O_m\}_{m\in\mathcal{M}}$ of the object in different modalities $\mathcal{M}$ and are defined as

$$\mathcal{T} = (\{O_m\}_{m\in\mathcal{M}}, \mathcal{P}) \tag{3.3}$$

Where $\mathcal{P}$ is a list of pairs (r, m) made up of the positions r of a specific feature in the modality m. Templates are generated by taking reference images of the object from different viewpoints around the object. The authors of [28] used about 2000 templates per object for matching. The ground truth pose for each template is found using markers which can be robustly detected. At test time the input image is divided up into small regions which are matched with the template features. Similar features are found using the following similarity measure.

$$\mathcal{E}(\{\mathcal{I}_m\}_{m\in\mathcal{M}}, \mathcal{T}, c) = \sum_{(r,m)\in\mathcal{P}} \left(\max_{t\in\mathcal{R}(c+r)} f_m((O_m(r), \mathcal{I}_m(t)) \right) \tag{3.4}$$

Where $\mathcal{R}(c + r)$ defines the neighborhood of a given size centered on location $(c + r)$ in the input image and f_m is the similarity score for modality m between the reference image and the input image. This means that the local neighborhood around a template feature at position r is aligned with the local features in the input image at location t.

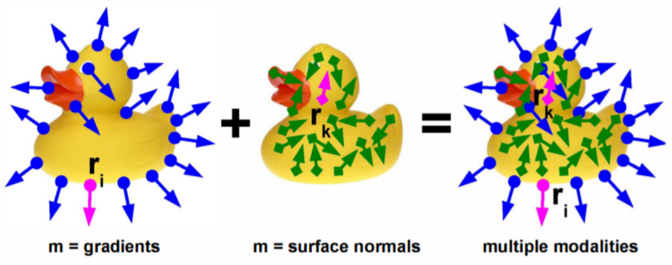


Figure 3.3: Example object with different modalities. Figure from [28].

The LINEMOD detection scheme was further improved, and fully automated, by the work presented in [29]. The ability to train templates directly from sampled viewpoints on 3D models of the objects was introduced. This significantly improved both the coverage and acquisition speed compared to manually sampling using an RGB-D sensor on physical objects. The resulting pose estimation pipeline is compared directly to the one developed by Drost et al. [17]. Scores are given as a percentage of correctly predicted poses compared with the ground truth pose for the object. On the dataset used, the authors report that their method outperformed the one of Drost et al. by an average of 17.7 percent on accuracy, while being an order of magnitude faster at test time.

In an effort to reduce the computational complexity of template based methods utilizing sliding windows such as LINEMOD, at test time, a cascade style evaluation of image locations is employed in [32].

3.3 Deep Learning Based Approaches

Pose estimation methods based on deep learning prior to 2018 were shown by Hodaň et al. [31] to be promising, but lacking in both accuracy and robustness compared to geometry and template based approaches. The reviewed methods were generally based on neural networks learning object features, primarily working on 2D intensity images, with some utilizing depth data for further pose refinement. Recent improvements upon these techniques and further development on approaches implemented based on geometric deep learning utilizing full 3D point clouds for pose estimation show promising results, improving accuracy, robustness and speed at test time.

3.3.1 Decision Forests

The problem of 6DoF pose estimation of textured and textureless objects is approached in [10] by introducing a joint dense 3D object coordinate predictor and object class labeler. To this end a decision forest is used to classify each pixel in an RGB-D image resulting in a prediction on which of the trained objects the pixel belongs to, and its location on the object known as object coordinate. Each predicted object coordinate corresponds to a 3D point on the 3D model of the object. Pose predictions are made using a RANSAC optimization scheme which samples sets of correspondences. The pose representing the best match is then iteratively refined to ensure proper alignment of the poses.

During training RGB-D features are learned from manually segmented images of the test objects. Pixel regions are sampled from this segmented image, and the forest is trained using ground truth object points. The pose estimation problem is formulated as an energy optimization problem, this minimization problem is used both during training and at test time for detection and pose refinement. Reported experimental results show that their method performed slightly better than the template based LINEMOD approach on the dataset from [28] while being more robust under difficult lighting conditions.

3.3.2 PoseCNN

Introduced in [70], PoseCNN is a Convolutional Neural Network (CNN) for 6DoF pose estimation using RGB images. The network is trained end-to-end, and the pose estimation pipeline consists of three main parts. A CNN performs semantic segmentation, labeling each pixel in the input image according to the object it belongs to. This results in a segmented image where each pixel is either labeled as part of an object of interest or as part of the background. The segmented pixel labels are then used to predict the translation from the camera center to the center of the object. Using a regression network, the the 2D pixel coordinates for each pixel on an object and it's distance from the camera is found. A Hough voting scheme is used to determine the pixel coordinate of the object's center point. A pin-hole camera model (Section 2.4.1) is then used to retrieve the position of the object in the camera coordinate system. Orientation is estimated for each object in the image using a regression network. Input to this network is the output features from the feature extraction part of the segmentation network, cropped by region of interest bounding boxes. The regression network consists of three fully connected layers which output a predicted unit quaternion representing 3D orientation.

The method for estimating the pixel coordinates of the object center improves the robustness towards occlusion, meaning the network is able to determine the object center even when it is not visible. The authors introduced a new loss function which does not penalize the orientation predictor for regressing to a valid orientation differing from the ground truth orientation in cases of symmetric objects. These features resulted in a pose estimation pipeline which is robust against both occlusion and self-symmetric objects. PoseCNN was evaluated on the Occluded-LINEMOD dataset [29]. The performance of the network was rather poor when utilizing the pipeline as a stand-alone pose estimator. However, when paired with an ICP refinement step the method outperformed both the implementations by Hinterstoisser et al. [29] and Brachmann et al. [10]. This final refinement step does require the accurate acquisition of depth imaging data in addition to the RGB images.

3.3.3 DeepIM

DeepIM or Deep Iterative Matching as introduced by Li et al. in [38] aim to utilize RGB images to estimate and refine the pose of an object in a cluttered scene. Their pipeline uses several convolutional neural networks for image pre-processing and pose refinement. The concept which is implemented in their approach is to use a trained neural network to iteratively refine the predicted object pose, given an initial pose estimate. In each iteration a rendered representation of the object's 3D model is compared with a segmented crop of the object in the input image. The network then predicts the difference in pose between the pose of the rendered object and the object in the image, this is used to update the pose estimate which is then fed to the next iteration. The conceptual pipeline is shown in Figure 3.4.

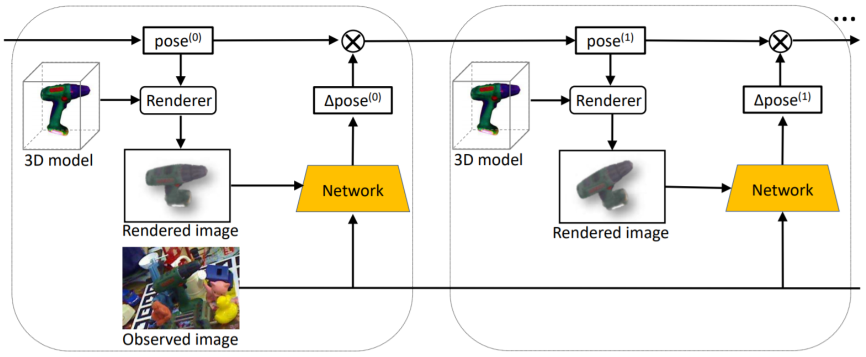


Figure 3.4: DeepIM pose refinement pipeline. Figure from [38].

The input image is cropped by using a pre-processing network to extract only the area of the image containing the object of interest. The authors experimented with two object detection networks, PoseCNN [70], and Faster R-CNN [55]. These networks output a predicted bounding box for the object, which was used for cropping. PoseCNN also outputs a predicted pose for the object, Faster R-CNN was modified by the authors to also output an estimate for the object orientation, while the center of the object bounding box was used to estimate 3D position. This was however less accurate than the pose estimate from PoseCNN. It is noted that the DeepIM pipeline performed similarly when using either pre-process network. Overall performance of this RGB only method is reported to be similar to approaches which use depth data for pose refinement, such as the use of ICP in combination with PoseCNN, while improving processing time.

3.3.4 Dense Fusion

Wang et al. [68] combines the power of convolutional neural networks for feature extraction and segmentation in the Euclidean domain, i.e. on 2D images, with the application of emerging developments within geometric deep learning [54] to encode geometric features directly from unordered 3D point sets. This new approach to feature extraction from point clouds differ from approaches which utilize learning on depth data represented as intensity images, such as in [18] where RGB and depth images are used to learn robotic grasping using two convolutional neural networks trained end-to-end on synthetic depth data. Instead geometric deep learning allow for the application of deep neural network models on non Euclidean domains such as graphs or manifolds [11]. A point cloud representing a 3D surface is a type of discrete manifold, which lends itself well to these techniques.

In Dense Fusion the object pixel segmentation and learned image features are combined with the segmented point cloud and extracted geometric features to attain dense correspondences between 3D point features and image features. Correspondences are found on a per pixel level by matching a point in the point cloud with a single pixel using a pin-hole camera projection based on known camera parameters. These combined features serve as the input to a network which learns to predict the pose of an object based on the densely fused local features as well as global features created from all local features.

The pose predictor is trained using a loss function based on the average absolute distance between sampled points on the object in the ground truth pose, and points on the surface of the object transformed by the predicted pose. Since one pose estimate is given per densely fused feature, special care is given to penalize predictions with a low confidence score.

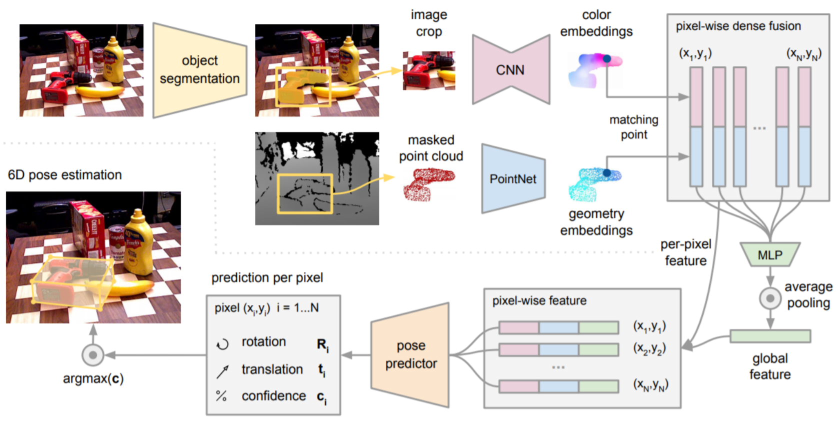


Figure 3.5: Overview of the Dense Fusion pipeline. Figure from [68].

While the accuracy of the predicted poses are shown to be 3.5% higher than that of PoseCNN + ICP refinement, on the YCB-Video dataset [72], it is clear that some form of pose refinement should be used to maximize the accuracy of the pipeline. However, in a similar vein to [38] the authors of [68] see the use of ICP based algorithms as having a severe detrimental effect on the possessing time of their pipeline. To this end a stand-alone refinement network utilizing densely fused features is proposed. The pose is refined by iteratively reducing the residual pose between the current segmented object point cloud and the transformed model point cloud. This refinement network can be trained at the same time as the main pose estimation network, but can only be introduced after the main network has converged due to poor learning performance on noisy input pose estimates. Overall performance of the Dense Fusion pipeline can be summarized as outperforming state-of-the-art pose estimation pipelines, both based on RGB and RGB-D data, by several percentage points on widely used datasets [28, 72]. It is also noted that the per frame processing time is 200 times faster than that of PoseCNN+ICP.

3.4 System Integration Concerns

For application of object pose estimators in automation of industrial component handling tasks, the production lead time and process adaptability may be considered as important as accuracy or frame processing time. Especially for robotic bin-picking systems deployed for small batch size production runs, or in production environments where low turnover times is an important consideration, the ability to quickly deploy object pose estimation systems for new components is important. If the pose estimation approach at hand requires a large amount of training data or lengthy supervised training it may not be cost effective despite boasting state-of-the-art performance at test time. These factors must be considered before implementing methods or systems for object pose estimation in industrial manufacturing applications.

For this reason geometry and template based approaches, which in many cases only require a 3D model of the object, lend themselves well to this kind of application. These methods "learn" matching features in a matter of seconds compared to the hours or even days of training required for deep learning based approaches. In addition most deep learning based approaches require very large datasets with annotated object poses or pixel masks for each frame used for learning. This necessitates a time-consuming process of dataset creation before new objects can be included in the system. The process of dataset generation can however, in some cases, be automated using physics simulators and rendering engines for the creation of synthetic datasets, as seen in [18]. Further developments on the use of synthetically generated training data as well as development of pipelines which can predict poses of unseen objects, is likely an important step in reducing lead times for the deployment of deep learning based pose estimators.

There are of course other considerations which impact the effectiveness of each approach. For instance, geometry and template based methods have been shown to be less robust in heavily occluded scenes. Under similar conditions deep learning based methods are shown to perform comparatively better. Another consideration is frame processing time. For bin-picking operations picking objects from a stationary bin, the longer processing time of geometry and template based approaches may provide adequate performance if it does not impact the overall cycle time to severely. In another application, for instance if a robot were to pick objects from a moving conveyor belt, the vastly faster frame processing time of deep learning based approaches may be necessary. Real time pose estimation and tracking of moving objects is likely outside the capabilities of geometry and template based approaches. It does however seem feasible for deep learning based approaches, given the current state of today's research. Especially when considering that current research on deep learning on non Euclidean domains are in relatively early stages compared with the more mature methods employed in convolutional neural networks on 2D images.

3.5 Hand Eye Calibrations Using Point Clouds

In order for a robotic manipulator to interact with its environment using vision systems, the pose of the camera relative to the robot base frame (eye-in-base, Figure 3.6b) or relative the robot end-effector frame (eye-in-hand, Figure 3.6a) must be found. The hand-eye calibration is used to transform the estimated pose of an object in the camera frame to obtain the pose of this object in the robot base-frame. In this section the kinematics of the hand eye calibration problem will be presented. A solution to the hand eye calibration problem will be derived following the method in [51], in addition a method utilising point clouds for hand-eye calibration is presented. The remainder of this section is based on the delivery in the specialization project in the course TPK4560 at NTNU [9].

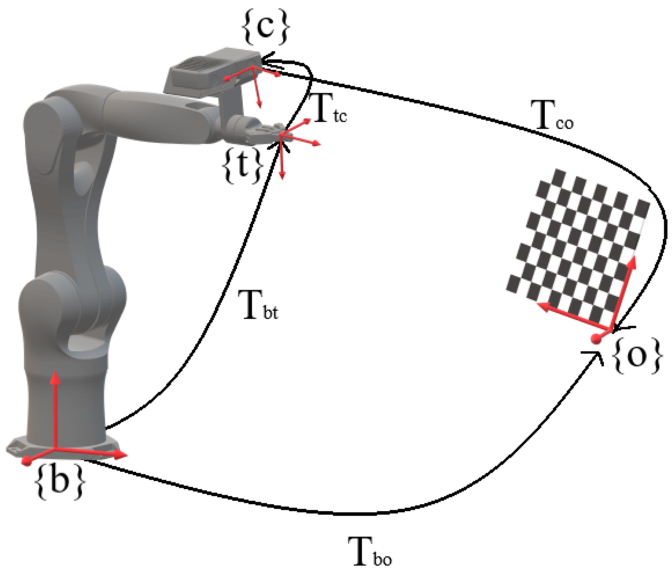
Work on solving the hand-eye calibration problem began in the late 1980s. At that time digital cameras were becoming good enough to be utilized in computer vision tasks for robotics. The hand-eye calibration problem was first formulated as a system of homogeneous matrix equations on the form $AX = XB$ where A represents robot motion and B represents camera motion. Shiu and Ahmad [59] found a closed form solution to this problem which satisfied the conditions for uniqueness. The determination of orientation was solved as the solution to a set of eight linear equations with four unknowns. Tsai and Lenz [65] proposed a different solution to the same problem formulation. They solved the rotational part of the calibration equation by represented the rotation by a unit unit vector $\boldsymbol{k_x}$ and an angle θ_x. The solution is formulated in a way that it can yield an exact closed form solution when two pose pairs are used, or a least squares solution if more pose pairs are used in the presence of noise. More pose pairs are used to help mitigate the effect of noisy measurements. Their solution is tested on both generated noisy data and on real robot/camera pose pairs. The camera positions are estimated using the extrinsic calibration methods introduced in [64]. Another solution which also generalizes to a linear least squares solution when noise is present is proposed by Park and Martin [51]. All solutions mentioned this far have proposed a separable solution to the problem by decoupling the rotation and translation parts of the formulated system of equations. First the solution for the rotational part of the hand-eye calibration matrix is found, then this is used to solve a set of vector equations to find the translation part. This is a good idea because it yields a relatively simple numerical optimization problem, but the linearization of the rotation problem can become ill conditioned in the presence of noise. Some of these issues were mitigated in the solution proposed by Horaud and Fadi [33]. Here, a formulation to the hand-eye calibration problem on the form of $MY = M'YB$ where the 3×4 perspective matrices M and M' of the camera in two different positions are used in stead of an explicit formulation of the extrinsic and intrinsic parameters. Two different solutions to this new formulation is presented. One closed form solution using linear least squares for optimization, and a non-linear solution using the Levenberg-Marquardt method to optimize the hand-eye transformation matrix. The second solution mitigates errors associated with decoupling rotation and translation, by optimizing both simultaneously.

The authors of [34] use time-of-flight and structured light 3D scanners to find the extrinsic parameters of the sensor by matching a the pose of a 3D model with captured 3D data. This was then used for hand-eye calibration using the method introduced by Tsai and Lenz. Their findings suggest that using their method of finding the extrinsic parameters of the cameras resulted in slightly more accurate hand-eye calibration results compared with the 2D image based method of extrinsic calibration. The 3D method did however require a more labour intensive process for dataset gathering, and was significantly more computationally expensive than the 2D image methods.

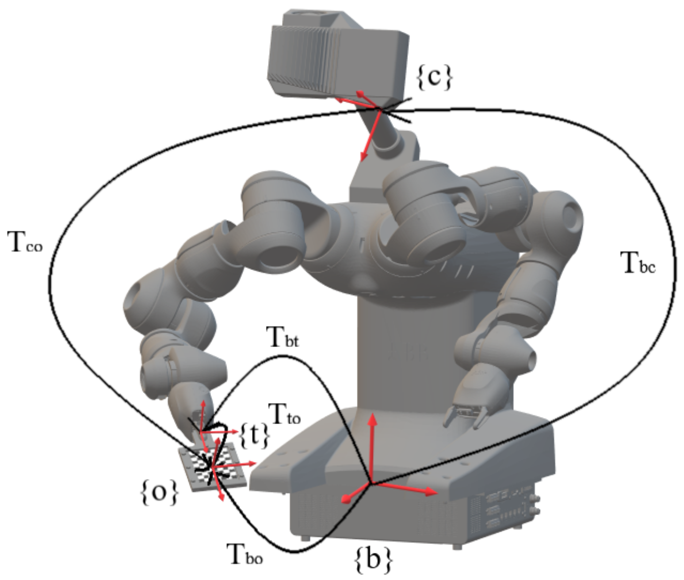
3.6 Kinematics of the Hand-Eye Calibration Problem

Given a eye-in-hand robot-camera configuration as shown in Figure 3.6a, with the following coordinate frames defined: robot base frame $\{b\}$, robot end-effector/tool frame $\{t\}$, a camera frame $\{c\}$ and an object frame $\{o\}$, the objective is to determine the pose of the object frame in the coordinates of the robot base frame. The transformation from the $\{b\}$ frame to the $\{t\}$ frame, T_{bt}, is found by using the forward kinematics of the robot manipulator, and the transformation from the $\{c\}$ frame to the $\{o\}$ frame, T_{co}, is found by a pose estimation algorithm. The transformation from the $\{t\}$ frame to the $\{c\}$ frame, T_{tc}, is calculated by hand-eye calibration. The notation $X = T_{tc}$ is often used for the unknown transformation between $\{t\}$ and $\{c\}$. Having determined all these transformations the pose of the object in the robot base frame, T_{bo} can be calculated by

$$T_{bo} = T_{bt}T_{tc}T_{to} \tag{3.5}$$



(a) Eye-in-hand calibration



(b) Eye-in-base calibration

Figure 3.6: Calibration setups, (a) camera mounted rigidly in relation to the end-effector frame, and (b) camera mounted rigidly in relation to the robot base frame.

To find the unknown camera calibration X, pairs of robot-camera configurations are used, as shown in Figure 3.7. Each pose pair consists of two robot poses and two camera poses. In the first configuration of the pose pair, the pose of the tool frame in $\{b\}$ and the pose of the object frame in $\{c\}$ is denoted A_{1a} and B_{1a} respectively. In the second configuration of the pose pair the poses are denoted A_{1b} and B_{1b}. The convention used is that the capital letter is used to distinguish between tool and object pose, the number is used to tell the pose pairs apart, and the lower case letters are used to differentiate between the first and second configuration in a specific pose pair.

Since the pose of the object in the robot base frame, T_{bo}, is the same for all pose configurations we can insert the pose pairs in to Equation (3.5) to get the following:

$$A_{1a} X B_{1a} = A_{1b} X B_{1b} \tag{3.6}$$

Pre-multiplying both sides of Equation (3.6) with A_{1b}^{-1} and post-multiplying both sides with B_{1b}^{-1} yields Equations (3.7) and (3.8).

$$A_{1b}^{-1} A_{1a} X = X B_{1b} B_{1a}^{-1} \tag{3.7}$$

$$A_1 X = X B_1 \tag{3.8}$$

Where A_1, B_1 are the relative transformations between the two poses in pose pair 1, for the tool and camera respectively. These transformations are shown as A and B in Figure 3.7. In order to find a unique solution for X at least two pose pairs must be used. To reduce the impact of noisy measurements it is usual to use several pose pairs, yielding a set of n equations.

$$\begin{aligned} A_1 X &= X B_1 \\ A_2 X &= X B_2 \\ A_3 X &= X B_3 \\ &\dots \\ A_{n-1} X &= X B_{n-1} \\ A_n X &= X B_n \end{aligned} \tag{3.9}$$

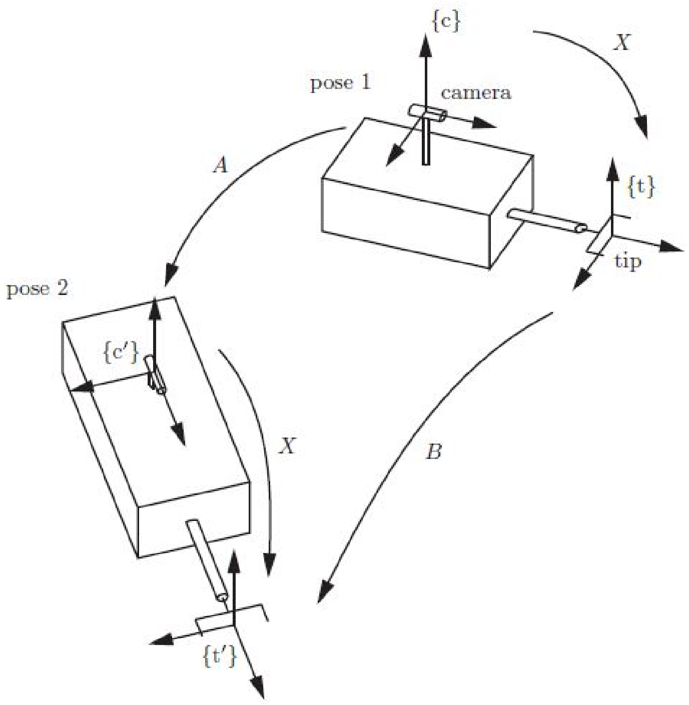


Figure 3.7: A camera mounted rigidly close to the robot end-effector. Figure from [39].

Derivation of the set of equations used in calibration of eye-in-base robot-camera configurations as in Figure 3.6b are only slightly different from the one used for eye-in-hand. Referencing Figure 3.6b the unknown transformation to be estimated is T_{bc}, herafter denoted X, from base frame to camera frame. The constant transformation T_{to} from the tool frame to the object frame is used to determine the lhs. and the rhs. of Equation (3.6). This constant transformation can be expressed as

$$T_{to} = T_{bt_i}^{-1} X T_{co_i} \tag{3.10}$$

for each configuration i. This is used to formulate the problem as a set of equations on the form of $AX = XB$, where A is the relative motion of the robot end-effector and B is the relative motion of the calibration object rigidly mounted to the end-effector.

3.7 A Solution to the Hand-Eye Calibration Problem

The derivation of the least squares solution to the set of equations presented in Equation (3.9), follows the method developed by Park and Martin[51]. This solution splits the problem of solving for the optimal orientation and the optimal translation between the reference frame and the camera frame into to separate problems. First the optimal rotation problem on $SO(3)$ is solved, this is then used to find the optimal translation. The combined result is the optimal transformation on $SE(3)$.

The matrix form of Equation (3.8) is Equation (3.11), which can be written as a pair of equations for rotation and translation (Equations (3.12) and (3.13)).

$$\begin{bmatrix} R_a & \boldsymbol{t}_a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_x & \boldsymbol{t}_x \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_x & \boldsymbol{t}_x \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_b & \boldsymbol{t}_b \\ 0 & 1 \end{bmatrix} \tag{3.11}$$

$$R_a R_x = R_x R_b \tag{3.12}$$

$$R_a \boldsymbol{t}_x + \boldsymbol{t}_a = R_x \boldsymbol{t}_b + \boldsymbol{t}_x \tag{3.13}$$

The best fit solution for the unknown rotation R_x between frame $\{t\}$ and frame $\{c\}$ minimizes η_1.

$$\eta_1 = \sum_{i=1}^{n} \|R_x log(R_{bi}) - log(R_{ai})\|^2 = \sum_{i=1}^{n} \|R_x(\theta_b \boldsymbol{k}_b)_i - (\theta_a \boldsymbol{k}_a)_i\|^2 \tag{3.14}$$

This minimization problem is equivalent to solving for the R_x which maximizes

$$\text{trace}(R_x K_b K_a^T) \tag{3.15}$$

Where $K_a = [(\theta_a \boldsymbol{k}_a)_1, ..., (\theta_a \boldsymbol{k}_a)_n] \in \mathbb{R}^{3\times n}$ and $K_b = [(\theta_b \boldsymbol{k}_b)_1, ..., (\theta_b \boldsymbol{k}_b)_n] \in \mathbb{R}^{3\times n}$. The optimal R_x is found by the singular value decomposition (SVD) of the matrix product $K_b K_a^T$ and forming R_x as in Equation (3.17), where S the Umeyama correction Equation (2.64) ensuring that R_x is a member of $SO(3)$.

$$U \Sigma V^T = \text{svd}(K_b K_a^T) \tag{3.16}$$

$$R_x = V S U^T \tag{3.17}$$

The optimal translation is found as the vector t_x which minimizes η_2.

$$\eta_2 = \sum_{i=1}^{n} \left\| (R_{ai} - I^{3\times3})t_x - R_x t_{bi} + t_{ai} \right\|^2 \tag{3.18}$$

The solution to this minimization problem is formulated as the least square solution to the matrix equation

$$Ct_x = d \tag{3.19}$$

$$C = \begin{bmatrix} R_{a1} - I^{3\times3} \\ R_{a2} - I^{3\times3} \\ ... \\ R_{an} - I^{3\times3} \end{bmatrix}, \quad d = \begin{bmatrix} R_x t_{b1} + t_{a1} \\ R_x t_{b2} + t_{a2} \\ ... \\ R_x t_{bn} + t_{an} \end{bmatrix} \tag{3.20}$$

The optimal translation is then found as

$$\boldsymbol{t}_x = C^{\dagger} d \tag{3.21}$$

Where $C^{\dagger}$ is the pseudo-inverse of C.

The resulting hand eye calibration matrix representing the tool-to-camera or base-to-camera transformation is

$$X = \begin{bmatrix} R_x & \boldsymbol{t}_x \\ 0 & 1 \end{bmatrix} \tag{3.22}$$

3.8 Procedure for Hand-Eye Calibration

The basic procedure for hand-eye calibration consists of the following steps.

1. Create a calibration dataset consisting of robot end-effector poses and images taken of the calibration object in different robot configurations.
2. Conduct extrinsic camera calibration for each robot configuration in order to find the pose of the calibration object in the camera frame for each image.
3. Create pose pairs and calculate relative robot and camera motion or robot and calibration object motion for each pose pair to obtain a set of equations as in Equation (3.9).
4. Calculate the best fit transformation between the robot end-effector frame and the camera frame by solving this set of equations.

3.9 Hand-Eye Calibration Using Point Clouds

To make use of the 3D point cloud data which generated by depth sensors, a method for extrinsic camera calibration utilizing point cloud data has been developed. First the corners of the calibration chessboard are found in the RGB image using the OpenCV python method `findChessboardCorners()`. This function call returns the sub-pixel coordinates of the internal corners on the chessboard. From these it would be possible to extract the 3D data directly, but since chessboard corners are high color contrast areas, the 3D data at these points is generally noisier than the data which corresponds to smoother gradient image regions. Instead the center point of each chessboard square is considered. The sub-pixel coordinates of these points are found as the point of intersection of each square's diagonals. The 3D point cloud from the Zivid One 3d camera is arranged in a $1920 \times 1200 \times 3$ matrix where points can be extracted using integer indices, e.g [202, 560]. Since the center point pixel-coordinates of each square generally will be a tuple of floats, e.g (202.56, 560.23), the 3D point corresponding to these pixel coordinates cannot be extracted directly. The four 3D points corresponding to the integer indices around the center point, (202, 560), (203, 560), (202, 561), (203, 261), are linearly interpolated to find the 3D coordinates of the center point in the camera coordinate system. This process is carried out for every center point on the calibration board, and for every image taken, to obtain a set of reliably placed 3D points on the calibration object relative to the camera.

For extrinsic camera calibration, the pose of the chessboard in the camera frame is found by fitting a plane to the 3D center-points, where the normal vector of the plane defines the z-axis of the chessboard coordinate frame. The x-axis is found by fitting a line to the 3D points along the long axis of the board, this vector is then projected into the plane orthogonal with the z-axis. The y-axis is found as the cross product $y = z \times x$. Each axis is then normalized to obtain a set of unit coordinate axes forming a basis in $\mathbb{R}^3$. This coordinate system is affixed to the centroid (Equation (2.58)) of the point cloud. When using this method of finding the extrinsic calibration of the camera for hand-eye calibration, the calibration method is referenced as the "3D plane fit method".

Since the formulation of the hand-eye calibration problem does not require the explicit pose of the camera in each configuration, only the relative motion of the camera or chessboard between two configurations, the transformation between the point clouds obtained in each configuration can be used instead. When using this method for hand-eye calibration it is referenced as the "3D correspondence method" or simply as the "3D method". The 3D plane fit method and the 3D method are collectively referred to as the "3D point cloud methods".

3.9.1 Calculation of Calibration Error

To evaluate the error of the different hand-eye calibration methods due to noise, the mean deviation in pose from the average estimated calibration object pose is considered. For eye-in-hand calibration the pose is referenced to the robot base frame, for eye-in-base calibration the pose is referenced to the end-effector frame. In both cases the constant transformation used to derive Equation (3.6) and Equation (3.10) is considered.

For error calculation on hand-eye calibration using OpenCV extrinsic calibration and the 3D plane fit method the pose of the calibration object is calculated for each robot configuration. This is done using the robot pose, the hand eye calibration and the extrinsic camera calibration, Equation (3.5). From this the mean rotation vector and the mean translation vector is calculated and used as an estimate for the ground truth pose of the calibration object. The error for each object pose estimation is then calculated as the transformation between the ground truth pose and the estimated object pose.

$$(T_{error})_i = (T_{bo})^{-1}_{GT}(T_{bo})_i \tag{3.23}$$

Where $(T_{error})_i$ is the estimation error for robot configuration i, $(T_{bo})_{GT}$ is the estimated ground truth pose, and $(T_{bo})_i$ is the pose estimation of the calibration object using configuration i. The hand eye calibration error is represented as translation error and rotation error. Translation error is calculated as the mean norm of the translation error vectors for all the estimates (Equation (3.24)), rotation error is calculated as the mean of the absolute value of the rotation angle error for all the estimates (Equation (3.25)). The angle of rotation, θ_{err}, is calculated using Equation (2.13).

$$t_{err} = \frac{1}{n}\sum_{i=0}^{n} ||(t_{err})_i|| \tag{3.24}$$

$$\theta_{err} = \frac{1}{n}\sum_{i=0}^{n} ||(\theta_{err})_i|| \tag{3.25}$$

For the 3D correspondence method, the pose of the calibration object is not explicitly known. The calibration error is therefore calculated slightly differently, while using the same metrics. The position of each calibration point on the calibration object is estimated using Equation (3.26) for each configuration i.

$$(P^b_o)_i = (T_{bt})_i(T_{tc})(P^c_o)_i \tag{3.26}$$

where $P_o^b \in \mathbb{R}^{4 \times n}$ is the homogeneous representation of the object point cloud in the robot base coordinate system and $P_o^c \in \mathbb{R}^{4 \times n}$ is the position of each calibration point in the camera coordinate system. An estimated ground truth position for each calibration point is calculated as the mean position from all the estimates $[(P_o^b)_0, (P_o^b)_1, (P_o^b)_2, ..., (P_o^b)_n]$. The error transformation is then calculated for each object pose estimation using point correspondences between the ground truth point cloud $(P_o^b)_{GT}$ and the calibration point cloud P_o^b for each configuration i. The translation and rotation error is then calculated using Equations (3.24) and (3.25).

Chapter 4

Motion Planning

In traditional robotized automation, obstacles are avoided by manual specification of the path of the manipulator. In order to avoid the cumbersome process of manually specifying paths around obstacles, motion planning algorithms can be employed. Motion planning algorithms convert high-level specifications of motions into specific descriptions of how to move, specifically solving the problem of determining the required translations and rotations to move a robot toward a goal while avoiding known obstacles in its workspace. For a robot to autonomously manipulate its environment, there is a need to endow the robot with capabilities to automatically plan its motions. In this chapter, the problem of motion and trajectory planning will be characterized. A sampling-based and a linear Cartesian motion planning pipeline using the MoveIt 2 Beta is outlined. The theoretical concepts presented in this chapter are based on descriptions and definitions from [37], [39] and [60].

4.1 Configuration Space and Operational Space

In order to establish convenient formulations, the notion of configuration space and operational space is introduced. The configuration space is given as the space containing all possible configurations a robot can assume. A configuration is a complete specification of every position of every point of the robot. For a robot with rigid links and n degrees of freedom, a minimum number of n real-valued coordinates are needed to represent a configuration [39]. In general, for a robot $\mathcal{B}_r$ made up of an open chain of r rigid links of known dimension and shape, connected by m revolute joints, the configuration is given by the vector of the relative orientations $\boldsymbol{q} = [q_1 \ ... \ q_m]^T$ between the links. The configuration space $\mathcal{C}$ for an m degree of freedom open chain is given by an m-dimensional manifold, however the shape of the manifold depends on the geometric properties of the kinematic chain. For instance, the geometric shape of the configuration space of an m degree of freedom open chain with a fixed base, will be of the shape of an m-dimensional torus [60]. Most robot manipulators have a large number of degrees of freedom, an thus the configuration space is often a high-dimensional space.

The operational space of a robot describes the space in which its manipulation is most easily specified, typically the three dimensional Cartesian space. While a configuration is a complete specification of a robot's pose, a pose is not necessarily a complete specification of a robots configuration. A pose in the operational space may be achievable by more than one configuration, especially if the robot features any degree of kinematic redundancy. In literature, operational space and task space is often used interchangeably.

4.2 Path and Trajectory

Furthermore it is useful to establish a clear distinction between a path and a trajectory. A path is a set of waypoints in the planning space which the manipulator has to follow while executing its motion [60]. The planning space can be either the configuration space or the operational space. By this definition the path is a purely geometric description of motion. A trajectory is a path coupled with a time-parameterization specifying the time at which each waypoint of the path is to be reached.

4.2.1 Trajectory Planning

Trajectory planning is the process of generating the reference inputs to the motion control that executes the planned path. The minimal requirement is that the planned trajectory can be completed by the manipulator. As the path only satisfies kinematic constraints, dynamic and differential constraints must be considered by the trajectory planner. Trajectory planning algorithms must therefore generate suitably smooth trajectories that adheres to the manipulators actuator limits. Trajectory planning can be done in the joint space and operational space.

Joint space trajectory planning algorithms generate a sequence of values for the joint variables which cause the manipulator to be moved from the starting configuration to the goal configuration while respecting any imposed constraints. This is typically done by generating a function $\boldsymbol{q}(t)$ interpolating the vector of joint variables at each waypoint. Two cases of joint space trajectory planning problems will be considered: point-to-point motion and motion through a sequence of points. Both problems will be generalized to the case of a single joint variable.

In the case of point-to-point motion, only the start and goal configuration and the travelling time are specified. This is the simplest case. In this situation a joint is to move from an initial joint configuration q_i to a final joint configuration q_f in a given time t_f. In this setting, the end-effector path is not considered, and the path itself contains only a starting configuration and a goal configuration. The *nth*-order polynomial function used to interpolate the start and goal can impose a specific joint velocity or joint acceleration profile, or optimize a performance index. For instance, the cubic polynomial $q(t) = a_3t^3 + a_2t^2 + a_1t + a_0$ can be used to produce a parabolic joint velocity profile and a linear joint acceleration profile. Another common approach is imposing a trapezoidal joint velocity profile. With this method a trapezoidal joint velocity profile is assigned, with a ramp-up, cruise and deceleration phase. In each phase the joint acceleration is constant: constant joint acceleration in the ramp-up phase, zero joint acceleration in the cruise phase and constant joint deceleration in the deceleration phase. The joint position, velocity and acceleration profile of a joint motion with trapezoidal joint velocity profile is illustrated in Figure 4.1.

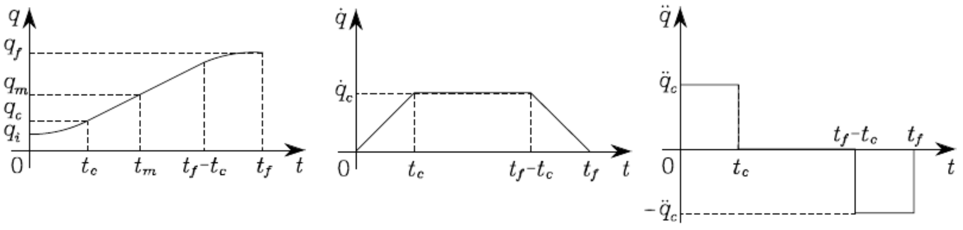
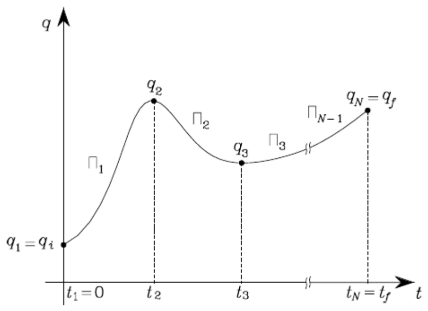


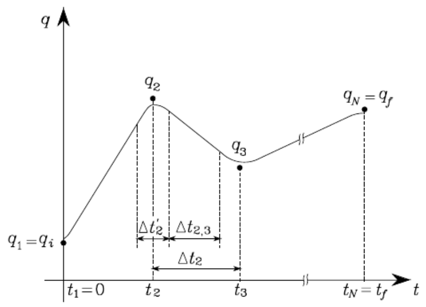
Figure 4.1: Characterization of a motion timing law with imposed trapezoidal joint velocity profile. Figure adapted from [60].

In most cases the path consists of more than two points. Intermediate points between the start and goal configuration allows more complex motion to be defined with better precision. Thus, it may be more convenient to define the trajectory as a motion through a sequence of points. However, adding points to the trajectory increases its polynomial complexity. N path points to be reached at specific time instances requires a $(N - 1)$-order polynomial to interpolate. Hence, for trajectories consisting of many intermediate path points, a high-order polynomial is required. Using a single high-order polynomial has many disadvantages: initial and final velocities cannot be specified, oscillatory behavior increase, changing a path point require the entire polynomial to be recalculated, and it is computationally complex to solve [60]. A high-order polynomial can however be avoided by using several low-order interpolating polynomials joined smoothly at waypoints. With this technique, to to be able to specify joint velocity and joint acceleration at waypoints, the functions interpolating the segments between the waypoints must be at least twice-differentiable, meaning the cubic polynomial is the lowest-order interpolating polynomial that can be used in this manner. This type of interpolation is typically referred to as spline interpolation, and is a popular interpolation technique in trajectory planning. A spline is a function defined piecewise by polynomials which are smoothly fitted at the joins, ensuring continuity of the function and its derivatives.

A simpler, alternative approach to the two formerly mentioned techniques is linear interpolation with blends. With this method, the trajectory is implemented using linear polynomials between the waypoints with blended joins between each linear segment. Blends are added to avoid discontinuity problems, typically using either a circular or cubic blend, depending on the differentiability requirements put on the trajectory. Due to the blends, the trajectory passes nearby, but do not reach any of the intermediate path points. A trajectory obtained by interpolating low-order polynomials can be seen in Figure 4.2a. Interpolation using linear segments and cubic joins can be seen in Figure 4.2b.



(a) Interpolating low-order polynomials



(b) Linear polynomials with cubic blends

Figure 4.2: Trajectories obtained through interpolating polynomials. Figures from [60].

In some cases, it may be desirable to plan the trajectory directly in the operational space of the robot. This can be done via either generating a time-sequence of operational space waypoints or by generating the time-parameterization of the parametric representation of the path. For both techniques, the waypoints of the trajectory in operational space must be transformed to the configuration space in real-time using an inverse kinematics algorithm. This adds significant computational complexity to the trajectory planning problem and puts an upper limit to the possible configuration space sampling frequency. Trajectories generated in operational space often result in a more 'natural' motion, however operational space trajectory planning is slower and the resulting joint motions are not necessarily smooth.

4.3 Defining the Motion Planning Problem

In order to address the motion planning problem in a precise manner, it is convenient to define the canonical problem. In this thesis, we will use the definition formulated in Siciliano et al. [60]. Given a robot $\mathcal{B}$, which consist of a single rigid body or a kinematic chain with either fixed or mobile base, that moves in a Euclidean workspace $\mathcal{W} = \mathbb{R}^N$, where $N = 2$ or 3. Let $\mathcal{O}_1, ..., \mathcal{O}_p$ be the obstacles, i.e. fixed rigid objects in $\mathcal{W}$. It is assumed that the geometry of $\mathcal{B}$, $\mathcal{O}_1, ..., \mathcal{O}_p$ is known, and that $\mathcal{B}$ is not subject to any kinematic constraints. The motion planning problem is then defined as the following: given the initial pose of the robot $\mathcal{B}$ in $\mathcal{W}$, if it exists, find a path, i.e. a continuous sequence of poses, which drives the robot $\mathcal{B}$ between the initial and goal pose while avoiding contacts between $\mathcal{B}$ and the obstacles $\mathcal{O}_1, ..., \mathcal{O}_p$. Report a failure if such a path does not exist.

By introducing the notion of the $\mathcal{C}$-*obstacle* region and the free configuration space $\mathcal{C}_{free}$ a more compact formulation can be reached. Let $\mathcal{C}$-*obstacle* denote the 'image' in $\mathcal{C}$ of an obstacle $\mathcal{O}_i$ present in $\mathcal{W}$. Then let $\mathcal{CO}_i$ denote the subset of configurations that cause a contact between the robot $\mathcal{B}$ and the obstacle $\mathcal{O}_i$. $\mathcal{CO}_i$ is then defined as:

$$\mathcal{CO}_i = \{\boldsymbol{q} \in \mathcal{C} : \mathcal{B}(\boldsymbol{q}) \cap \mathcal{O}_i \neq \emptyset\}\,. \tag{4.1}$$

Then, the $\mathcal{C}$-*obstacle* region is the union of all subsets $\mathcal{CO}_i$ that cause the robot $\mathcal{B}$ to be in contact with any obstacle $\mathcal{O}_1, ..., \mathcal{O}_p$. Meaning the the $\mathcal{C}$-*obstacle* region, $\mathcal{CO}$, is the union of $\mathcal{CO}_i,\ i = 1, ..., p$

$$\mathcal{CO} = \bigcup_{i=1}^{p} \mathcal{CO}_i \tag{4.2}$$

The free configuration space is then defined as $\mathcal{C}_{free} = \mathcal{C} - \mathcal{CO}$. While $\mathcal{C}$ is a connected space, due to the obstacles, $\mathcal{C}_{free}$ is not necessarily fully connected. With these definitions, we formulate the compact version of the canonical motion planning problem. Assume $\mathcal{B}$ is not subject to any kinematic constraints. The initial and goal pose of $\mathcal{B}$ is mapped to $\mathcal{C}$ as the initial configuration $\boldsymbol{q}_i$ and the goal configuration $\boldsymbol{q}_g$. Planning a collision free motion for the robot $\mathcal{B}$ means generating a path connecting $\boldsymbol{q}_i$ and $\boldsymbol{q}_g$, contained in single, connected subset of $\mathcal{C}_{free}$, and reporting a failure otherwise. The formulated definition assumes the robot to be the only object in motion in the workspace and that the placement and geometry of the obstacle are known a priori to the planning.

The formulated definition of the motion planning problem has reduced motion planning to an entirely geometrical path planning problem. This is useful as many methods solve the geometric path planning problem in one step, and apply trajectory planning to generate a feasible trajectory in another step.

4.4 Motion Planning Methods

There is no single planner suitable for every robot and every motion planning problem. Dependent on the task at hand and the robot to be used, different planning algorithms is typically to be preferred. There are several classes of planners, however, with our purely geometrical definition, it can be argued there are roughly three main categories: combinatorial methods, sampling-based methods and potential field methods. Combinatorial methods represent the connectivity of the free configuration space $\mathcal{C}_{free}$ by constructing a roadmap. A roadmap is a graph where each vertex represent a configuration and each edge a path between two configurations, containing enough paths to adequately describe the connectivity of $\mathcal{C}_{free}$ [37]. This requires an explicit computation of the $\mathcal{C}$*-obstacle* region $\mathcal{CO}$, which is a very complex computation. Combinatorial methods is mainly separated by how they construct the roadmap. After the roadmap is constructed, a graph search algorithm is deployed to find a feasible path between the initial configuration $\boldsymbol{q}_i$ and the goal configuration $\boldsymbol{q}_g$. Depending on the search algorithm used, different path properties can be prioritized, e.g. shortest path or greatest distance to obstacles. Combinatorial methods are complete, meaning a solution can be guaranteed to be found if it exists, and a failure will be reported otherwise. They are also multiple-query, as once the roadmap is constructed, and no changes in the environment occurs, the same roadmap can be used to solve multiple planning queries. Owing to the need for a explicit computation of $\mathcal{CO}$, combinatorial methods are unsuitable for high-dimensional problems. A large portion of combinatorial motion planning algorithms are dedicated to the planar mobile robot case, with $\mathcal{C} = \mathbb{R}^2$, $\mathcal{CO}_i$ is polygonal and the robot $\mathcal{B}$ is a polygon only capable of translation.

Sampling-based methods are a very popular set of methods for high dimensional planning problems. Sampling-based methods avoid the explicit calculation of $\mathcal{CO}$ by sampling the configuration space using either a deterministic or randomized sampling scheme. The methods typically build a roadmap incrementally by collision-checking sampled configurations using a separate collision checker, adding collision-free configurations to the roadmap and discarding those in collision [37]. By outsourcing collision checking to a separate module, the planners can be applied to a wider range of problems by tailoring the collision detector to the specific robot or application. Alternatively, a very efficient collision-detection algorithm can be used to achieve a fast planning. Owing to the probabilistic nature of the sampling-based planners they are probabilistically complete, meaning they can guarantee to eventually find a solution, if it exists, however they cannot detect an impossible planning request. Sampling-based methods can be both single-query and multiple-query. Multiple-query methods use a probabilistic approach to map the set of possible configurations $\mathcal{C}_{free}$. This map is used to process later queries at a lower computational expense. This requires $\mathcal{CO}$ to be static. Even using a probabilistic approach mapping of $\mathcal{C}_{free}$ is computationally very expensive.

Potential field methods are based on an entirely different approach. Instead of generating a roadmap, they construct a differentiable real-valued function U used to guide the motion of the robot. The potential function U is obtained as the superposition of an attractive potential to the goal and a repulsive potential from the $\mathcal{C}$-obstacles [60]. Planning is done incrementally: at each configuration $\boldsymbol{q}$ the next step is determined from the negative gradient $-\nabla U(\boldsymbol{q})$ of the potential, which indicates the most promising direction in the configuration space. However, this gradient-descent based approach do not guarantee a solution to be found, as the planning can get stuck in a local minima. Although there are techniques for handling local minima, potential field methods are generally not complete. However, unlike combinatorial methods and sampling-based methods, potential field methods are suitable for online planning. The formerly presented families of methods dependent on a priori knowledge of the geometrical characterization of the environment and plan the path before any motion is initiated. Online planning refers to planning where some of this information is gathered from sensors and used to incrementally generate the path during motion. Potential field methods plan in an incremental fashion, allowing new obstacle repulsive fields to be added between iterations in the planning. Potential field methods are suited to plan in high-dimensional spaces, and are inherently single-query.

4.5 MoveIt 2: ROS 2 Motion Planning Framework

The MoveIt motion planning framework is a set of open-source ROS packages offering capabilities for motion planning, manipulation, 3D perception, kinematics, control and navigation [12]. It was originally developed by Willow Garage as a generalized version of the `_navigation package` developed for the PR2 robot, with the first beta released in 2013 and the first official version, MoveIt 1.0, in 2019 [5]. In February 2020, the first beta for the ROS 2 version of MoveIt, MoveIt 2, was released. In the MoveIt 2 beta release, much of the core components of MoveIt 1.0 has been ported to ROS 2. Most notably missing is the MoveGroup high-level ROS API which gives access to MoveIt functionalities through ROS actions, services and topics. Instead, the `moveit_cpp` C++ high-level API is available. This is an advanced developer API which gives access to core MoveIt 2 functionality directly [43].

In agreement with our definition of the motion planning problem from Section 4.3, in MoveIt 2, the motion planners are setup to plan paths [42]. In the complete motion planning pipeline, the motion planner of choice is chained together with planning request adapters, which provide options for preprocessing of planning request, and postprocessing of the planned path. A typical planning pipeline consists of three stages: preprocessing, planning and postprocessing. In the preprocessing stage, an incoming planning request issued from user code is preprocessed in one ore more planning request adapters to ensure the request is valid and contains the required fields. In the planning stage, the motion planner plans a path according to the preprocessed motion planning request, using a dedicated collision checker module to ensure a collision free path is generated. In the postprocessing stage, the generated path is time-parameterized to create a workable trajectory.

4.5.1 OMPL-based Motion Planning Pipeline

In order for MoveIt 2 to function with different motion planners from different libraries, MoveIt 2, similar to the original MoveIt, integrates motion planners through a plugin interface. The beta release of MoveIt 2 used in this thesis integrates motion planners from the Open Motion Planning Library (OMPL) [63], and comes bundled with the OMPL motion planners. OMPL is an open-source C++ library of sampling-based motion planners. OMPL is centered around the planners and provides a uniform interface with clear mappings between abstract classes and theoretical concepts. OMPL therefore integrates nicely in complex systems. However, being constrained to only the planners, OMPL provide no means for visualization or collision checking. Collision checking is handled by a StateValidity checker which interfaces with an external collision checker to evaluate if a sampled state is collision-free. Most of the motion planners in OMPL are geometric path planners, considering only geometric and kinematic constraints.

In the OMPL-based motion planning pipeline in MoveIt 2, a motion planning request is constructed from a planning goal. A planning goal is constructed using a constraint message containing fields for joint constraints, position constraints, orientation constraints and visibility constraints. Joint constraints are used for state goal planning and position and orientation constraints for pose goal planning. Visibility constraints can be used to enforce visibility of a link in a defined volume representing the cameras field of view. Regardless of which constraints are used to define the goal, planning is performed in the configuration space. For a pose goal, the goal must therefore be transformed to a configuration before planning can start. This is performed behind the scenes by the kinematics plugin. For manipulators where several configurations can yield the same pose, the goal configuration realized can differ between planning requests to the same pose goal.

Preprocessing is performed on the given motion planning request using four planning request adapters: FixStartStateBounds, FixWorkspaceBounds, FixStartStateCollision and FixStartStatePathConstraints. The FixStartStateBounds adapter handles situations where the start state of the robot is slightly outside the specified joint limits. This may occur when the specified joint limits is slightly more constrained than the real joint limits. In this situation the adapter will modify the start state to be within the joint limits. The FixWorkspaceBounds adapter sets a default workspace of a 10m$\times$10m$\times$10m cube to be used for motion planning request when no workspace bounds are given. The FixStartStateCollision adapter will attempt to sample a new collision-free configuration of the robot if the start state is in collision. The amount it will be allowed to adjust the joint values of the start state is specified by an user defined value called the "`jiggle_factor`". Additionally, the start state will be checked to be within the path constraints given by the motion planning request. If needed, the FixStartStatePathConstraints adapter will attempt to plan a a path between the start configuration to a new location where the path constraint is obeyed, and set this to be the start state for planning.

When the motion planning request have been preprocessed, it is sent to the motion planner specified for the planning request. The RRT-Connect motion planner is the default planner of the OMPL-based motion planning pipeline in MoveIt 2. Due to MoveIt 2 currently missing functionality for changing the motion planner employed, RRT-Connect will be the motion planner employed in this thesis. RRT-Connect is a sampling-based motion planning algorithm first proposed by Kuffner and LaValle in [35], and is based on the principle of two Rapidly-Exploring Random Trees (RRT). RRT is a tree data structure which is incrementally extended using a randomized extension-procedure to be repeated at every iteration. At each iteration, a randomly selected configuration q_{rand} is sampled from $\mathcal{C}$. The tree is then searched with the nearest neighbour algorithm to find the configuration q_{near} already stored in the tree which is the closest to q_{rand}. Then q_{near} and q_{rand} are joined with a segment, and a new configuration q_{new} is found by traversing this segment a fixed distance ϵ from q_{near}. A collision check is performed on q_{new} by an external collision checker to ensure it is collision-free, i.e. to verify that $q_{new} \in \mathcal{C}_{free}$. If q_{new} is found to be collision-free, it is added to the tree as a vertex, together with the edge connecting q_{near} to q_{new}. The tree is extended in this fashion until there exist a path between the start configuration q_{start} and the goal configuration q_{goal}. RRTs are very efficient in exploring the configuration space, and the procedure for generating new candidate configuration is biased towards new, unvisited regions of $\mathcal{C}_{free}$. The extension-procedure is summarized in Algorithm 1 and illustrated in Figure 4.3.

Algorithm 1 EXTEND($\mathcal{T}, q$)

```
$q_{near} \leftarrow$ NEAREST_NEIGHBOR($q, \mathcal{T}$)
**if** NEW_CONFIG($q, q_{near}, q_{new}$ **then**
    $\mathcal{T}$.add_vertex($q_{new}$)
    $\mathcal{T}$.add_edge($q_{near}, q_{new}$)
    **if** $q_{new} = q$ **then**
        Return *Reached*
    **else**
        Return *Advanced*
Return *Trapped*
```

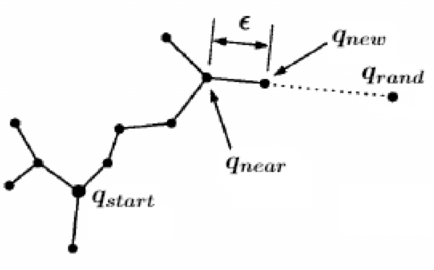


Figure 4.3: The EXTEND procedure. Figure from [35].

RRT-Connect is based on two ideas: growing two RRTs, $\mathcal{T}_a$ rooted at the start configuration q_{start} and $\mathcal{T}_b$ rooted at the goal configuration q_{goal}, and the Connect heuristic. The Connect heuristic is a greedy function that attempts to extend an RRT over a larger distance. Instead of extending the RRT in a single ϵ step, the Connect heuristic repeats the extension step until either q_{rand} or an obstacle is reached. RRT-Connect alternate between growing the two trees until they connect, at which point a solution is found. In each iteration of RRT-Connect, one tree is extended and an attempt to connect the trees is made. In the next iteration the roles are swapped. The extension is done using the normal RRT extension procedure listed in Algorithm 1, however the attempt at connecting the two threes is done using the CONNECT heuristic, during which the nearest vertex of the other three is attempted extended to the new vertex q_{rand}. The swapping of roles allow both trees to explore $\mathcal{C}_{free}$ while trying to establish a connection between them. The Connect function can be seen in Algorithm 2 the full RRT-Connect algorithm can bee seen in Algorithm 3.

Algorithm 2 CONNECT($\mathcal{T}, q$)

```
**repeat**
    $S \leftarrow$ EXTEND($\mathcal{T}, q$);
**until not** (S = *Advanced*)
Return $S$;
```

Algorithm 3 RRT-Connect(q_{start}, q_{goal})

```
$\mathcal{T}_a$.init(); $\mathcal{T}_b$.init();
**for** $k = 1$ **to** $K$ **do**
    $q_{rand} \leftarrow$ RANDOM_CONFIG();
    **if not** (EXTEND($\mathcal{T}_a, q_{rand}$) = *Trapped*) **then**
        **if** (CONNECT($\mathcal{T}_b, q_{new}$) = *Reached*) **then**
            Return PATH($\mathcal{T}_a, \mathcal{T}_b$);
        SWAP($\mathcal{T}_a, \mathcal{T}_b$);
Return *Failure*;
```

Sampled states from the planner are checked for collision with the planning scene, a representation of the world around the robot and the robot's state. The world representation in the planning scene can be modified in a real-time manner from sensor data, or predefined using geometric primitives and meshes. Collision is checked by collision check queries to the Flexible Collision Library (FCL) [50]. FCL is an open-source library that includes various techniques for efficient collision detection and proximity computation. The library represents objects with Bounding Volume Hierarchies (BVH) and uses hierarchical object representation and tree traversal techniques to speed up collision checking. A BVH is a tree data structure of geometric objects, where the objects are bounded by bounding volumes in a hierarchical manner. The bounding volumes form the internal nodes of tree, while the objects the leaves. All child node bounding volumes are enclosed by their parent node bounding volume. The root bounding volume enclose all bounding volumes of the tree. If a parent volume is found to have no overlap, the child volumes the parent do not need to be checked. The BVH presentation therefore removes the need to check for collision on every geometric primitive comprising a object, and thus saves a lot of computation. An example of a BVH can be seen in Figure 4.4.

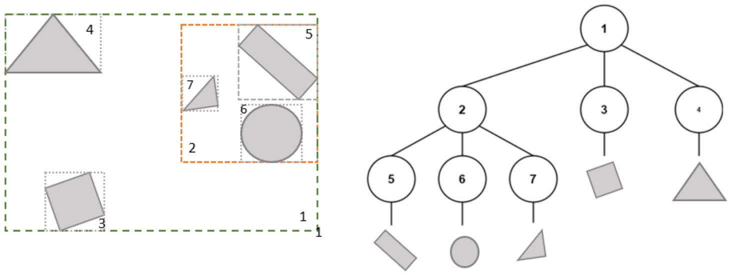


Figure 4.4: An example of a bounding volume hierarchy.

MoveIt 2 checks for collision using discrete collision check queries. A discrete collision check query is executed in three phases: first a self-collision check, then a broad-phase collision check, and finally a narrow-phase collision check. Self-collisions are checked with an unpadded version of the provided collision meshes of the robot. If passed, the the broad-phase check is started. In the broad-phase collision check, a sweep-and-prune algorithm is executed. The environment around the robot is modelled with axis-aligned bounding boxes in a BVH, and the algorithm traverse the hierarchy-trees searching for overlapping bounding boxes. Next, narrow-phase collision checking calculations are executed for the overlapping pairs of bounding boxes. These calculations are performed directly on the meshes of the objects in the bounding boxes, which are a far more computationally expensive than collision checking on bounding boxes. However, due to the use of BVHs, these calculations are kept to a minimum.

Collision checking is done in the Cartesian space. Therefore, sampled configurations are run through a forward kinematics algorithm to find the pose of the manipulator at that state before the robot's padded collision meshes are checked for overlaps the axis aligned bounding volumes of the obstacles. Additionally, the segments connecting the vertices of the path are also collision checked at intermittent points. Collision checking is a very computationally expensive operation and accounts for a large part of the computational expense of a motion planning request. To alleviate this, MoveIt 2 uses an AllowedCollisionMatrix datatype to store information about pairs of bodies that do not need to be collision checked. Objects which can never be in collision, or will always be in collision, for instance some links of the robot, may be encoded into the matrix. The AllowedCollision-Matrix can also be modified during runtime.

When a collision-free path is found, the path is postprocessed with a time-parameterization to transform it to an executable trajectory. In this pipeline, the adapter AddTimeOptimalParameterization is used. The adapter generates a smooth time-optimal trajectory within the given bounds on joint velocities and joint accelerations after the principle introduced in [36]. The adapter generates the trajectory in two steps: first a preprocessing step to make the found path twice-differentiable, next the time-optimal time-parameterization, where a trajectory which exactly follows the differentiable path within the given bounds on joint velocity and joint acceleration is produced.

RRT-Connect, and most sampling-based planners, produce paths in configuration space consisting of waypoints connected with straight line segments. At the waypoints, the path is not differentiable. For the algorithm to able to time-parameterize these paths directly, these connections must be be smoothed. This is done in a preprocessing step in the time-parameterization by adding circular blends around the waypoints. The blends are limited to not encompass more than half of the neighboring linear segments, and a maximum deviation from the original path is enforced. The time-parameterization is given in joint velocity and joint acceleration and computes a time-optimal trajectory where the manipulator is moving as close as possible to its velocity limits by applying either minimum or maximum acceleration. The trajectory is computed using a limit curve constructed from the minimum of the joint velocity and joint acceleration. This curve is found in the $(s, \dot{s})$ plane, where s denotes the travelled arc length.

After the time-parameterization, the pipeline has completed its processing of the motion planning request, and a viable, collision-free trajectory has been produced. A flowchart of the pipeline can be seen in Figure 4.5.

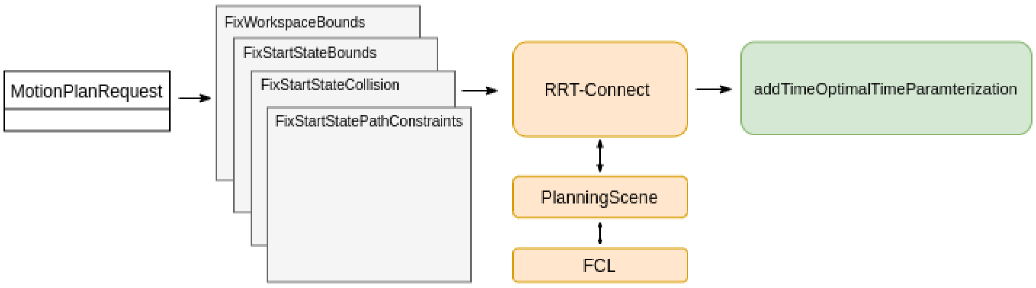


Figure 4.5: Flowchart of the OMPL-based MoveIt 2 motion planning pipeline.

4.5.2 Linear Cartesian Motion Planning Pipeline

While the OMPL-based pipeline presented in Section 4.5.1 will find a viable, collision-free trajectory, the trajectory can take any arbitrary path in the operational space as long as the constraints of the motion planning request is satisfied. In many applications, constraining the motion to a Cartesian motion primitive may be required, for instance in bin-picking, assembly or welding applications. There is however no option to constrain the trajectory to a a Cartesian motion primitive in the OMPL-based pipeline. Thus, a second linear Cartesian MoveIt 2 motion planning pipeline using MoveIt 2 resources is developed. The outlined linear Cartesian MoveIt 2 motion planning pipeline consists of tree steps: an interpolation step for finding configurations along a straight line toward the given pose goal, a collision check of the interpolated path and finally time-parameterization to obtain a trajectory.

The interpolation is done using the CartesianInterpolator of MoveIt 2. A pose goal is given to the interpolator together with limits on allowed maximum Cartesian distance between waypoints and maximum allowed joint value jumps in the state space. A set of configuration space waypoints forming a straight line between the start pose and goal pose in Cartesian space is produced. Configuration space waypoints are generated by finding the next waypoint in Cartesian space, and transforming this to the configuration space using an numerical inverse kinematics solver. When a path is found, the path is collision checked using FCL. This collision check procedure is done similarly to the OMPL-based motion planning pipeline. When deemed collision free, the path is time-parameterized. The time-parameterization is applied using the addIterativeSplineParameterization planning adapter. The adapter respects limits on joint velocity and joint accelerations.

The code-implementations of the MoveIt 2 motion planing pipelines is presented in Section 6.3.3.

Chapter 5

Estimation of External Forces

The measurement or estimation of external forces acting on a robot manipulator can be used for several applications. One application which is particularly useful for collaborative robots is the application of contact force monitoring for collision detection purposes. This can be used as a safety feature in robotics systems by immediately seizing motion upon the detection of a collision in order to reduce damage to the environment, the robot, or human workers interacting with the manipulator. Another useful application is the use of measurements, or estimations, of forces acting in the robot end-effector frame to control the robot. This type of force control can be especially useful for assembly operations involving components with tight tolerances or where a specified force or torque is to be applied to meet the required specifications in an assembly. Force control can also be used in combination with constraint based control where the robot is programmed based on the constraints of an assembly, e.g. the alignment of the axes of a hole and a cylinder. In this case force control can be used to detect errors in alignment based on the forces and moments resulting from contact and adjust the robot accordingly. To facilitate these applications we intend to implement a system for the sensor-less estimation of end-effector forces and moments, thus, the main purpose of this chapter is to present the methods used for such a purpose.

The use of force control rely on accurate measurements of forces and torques applied to the robot end-effector. These measurements are generally done with dedicated force-torque sensors attached to the end-effector. This type of sensor allows for direct measurement of the contact forces between the robot and the environment. Implementation of such a sensor may, however, not always be convenient or at all possible. Instances where the end-effector payload would be severely reduced due to the added weight of the sensor, or where the robot manufacturer does not provide a means of attaching such a sensor, inhibit their use. This is the case for the ABB YuMi which has a payload limit of 0.5 kg. In these cases a scheme for sensor-less estimation of contact forces based on the robot dynamics and applied actuator torques can be used.

5.1 Method for Estimation of External Forces

The model used for estimation of forces between the robot end-effector and the surrounding environment is based on the work of De Luca et al. [15, 14]. Their model is used for collision detection without the use of external sensors. A similar model is utilized by Nicolis et al. [45] and by Polverini et al. [53] in constraint based force control schemes carrying out tasks such as peg-in-hole assembly. The derivation of a method for estimation of contact forces on the end-effector frame follows.

Recall Equation (2.34) describing the inverse dynamics problem for a serial manipulator, derived using a Lagrangian approach. This equation is modified to also include the resulting motor torques $\boldsymbol{\tau_c}$ from contact forces between the end-effector and the environment.

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

The system Lagrangian $\mathcal{L} = \mathcal{K}(\boldsymbol{q}, \dot{\boldsymbol{q}}) - \mathcal{P}(\boldsymbol{q})$, where $\mathcal{K}$ and $\mathcal{P}$ are the kinetic and potential energy of the system respectively, is used to derive the system's generalized momentum $\boldsymbol{p}$. The elements of $\boldsymbol{p}$ are found from

$$p_i = \frac{\partial \mathcal{L}}{\partial \dot{q}_i} = \frac{\partial \mathcal{K}(\boldsymbol{q}, \dot{\boldsymbol{q}})}{\partial \dot{q}_i} - \frac{\partial \mathcal{P}(\boldsymbol{q})}{\partial \dot{q}_i} = \frac{1}{2}\dot{\boldsymbol{q}}^T \frac{\partial M(\boldsymbol{q})}{\partial \dot{q}_i}\dot{\boldsymbol{q}} \tag{5.2}$$

Define the generalized momentum for the system as

$$\boldsymbol{p} = M(\boldsymbol{q})\dot{\boldsymbol{q}} \tag{5.3}$$

with time derivative $\dot{\boldsymbol{p}} = \frac{d}{dt}\boldsymbol{p} = M(\boldsymbol{q})\ddot{\boldsymbol{q}}$, which satisfies

$$\dot{\boldsymbol{p}} = \boldsymbol{\tau} + \boldsymbol{\tau_c} - C(\boldsymbol{q}, \dot{\boldsymbol{q}}) - \boldsymbol{g}(\boldsymbol{q}) \tag{5.4}$$

The external torques are estimated by calculating the residual

$$\boldsymbol{r} = K_R\left[\boldsymbol{p} - \int (\boldsymbol{\tau} - C(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}} - \boldsymbol{g}(\boldsymbol{q}) + \boldsymbol{r})dt\right] \tag{5.5}$$

The residual satisfies

$$\dot{\boldsymbol{r}} = K_R(\boldsymbol{\tau_c} - \boldsymbol{r}) \tag{5.6}$$

The contact forces can then be calculated by establishing the relationship between external forces and the torques resulting from these forces.

Starting with the definition of power $P = \frac{W}{t} = F^T V$. For external forces acting on the end-effector we have

$$P_c = F_t^T \mathcal{V}_t \tag{5.7}$$

where F_t is the wrench of forces and torques acting on the end-effector frame and $\mathcal{V}_t$ is the end-effector twist. This same quantity can be calculated from the actuator torques resulting from the external wrench $\boldsymbol{\tau_c}$ and the joint-velocities.

$$P_j = \boldsymbol{\tau_c}^T \dot{\boldsymbol{q}} \tag{5.8}$$

From this we find the relationship between the contact forces and the resulting joint-torques.

$$\boldsymbol{\tau_c}^T \dot{\boldsymbol{q}} = F_t^T \mathcal{V}_t \tag{5.9}$$

$$\boldsymbol{\tau_c}^T \dot{\boldsymbol{q}} = F_t^T J(\boldsymbol{q}) \dot{\boldsymbol{q}} \tag{5.10}$$

$$\boldsymbol{\tau_c} = J^T(\boldsymbol{q}) F_t \tag{5.11}$$

Where $J(\boldsymbol{q})$ is the manipulator Jacobian defining the mapping $\mathcal{V} = J(\boldsymbol{q})\dot{\boldsymbol{q}}$. Using Equation (5.11) the external forces can be estimated from the residual calculated in Equation (5.5) as

$$F_t = J^T(\boldsymbol{q})^\dagger \boldsymbol{r} \tag{5.12}$$

Where $J^T(\boldsymbol{q}^\dagger)$ is the pseudo inverse of the manipulator Jacobian, for an $n \times m$ Jacobian matrix. If the Jacobian matrix is $n \times n$ it is invertable.

Chapter 6

System Description

The realized robot system integrates automatic motion planning, 3D computer vision and external force estimation. The system is a continuation of the work in [46], and comprises several hardware and software components. The implementation of, and the interaction between these components are described within this chapter.

6.1 Hardware

6.1.1 ABB YuMi

The robot used in this thesis is the collaborative, dual-arm industrial robot IRB 14000 YuMi, introduced by ABB in 2015 [1]. It was designed for assembly operations in manufacturing, specifically to meet the production needs of the consumer electronics industry [71]. Being a collaborative robot, YuMi is designed to operate safely alongside humans. YuMi's arms are therefore lightweight and padded, and the robot controller can stop the motors upon detected collision. The arms are equipped with seven revolute joints, providing seven degrees of freedom to each arm. This allows YuMi to maintain a larger workspace and reach more diverse poses than a traditional 6-axis industrial robot. Due to its low weight and table mountings, YuMi can be easily moved and redeployed. YuMi can be seen in Figure 6.1.

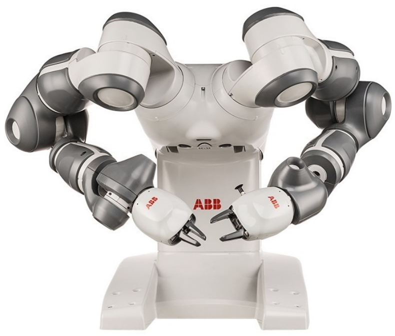
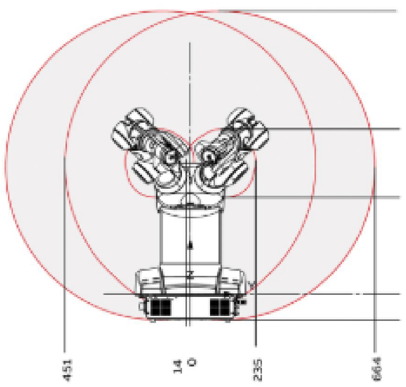


Figure 6.1: YuMi front facing.

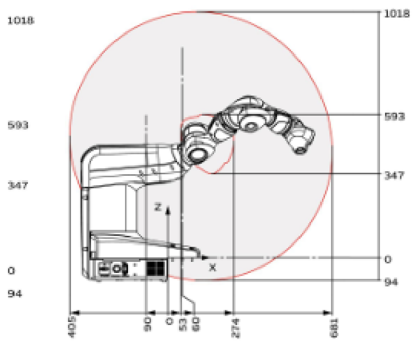
YuMi is fast and precise. Measured at the Tool Center Point (TCP), with no eternal loads, it is capable of Cartesian velocities of 1.5 m/s and a Cartesian acceleration of 11 m/s^2 while maintaining positional repeatability of 0.02 mm [71]. Being designed for assembly of consumer electronics, it has got a low payload of 0.5 kg. The arms have overlapping workspaces, meaning there is a volume in the workspace where both arms can reach the desired pose, giving YuMi flexibility in more complex, dual-arm manipulation tasks. A summary of properties from the data sheet are given in Table 6.1. A front and side view of its workspace can be seen in Figure 6.2.

Property	**Value**
Degrees of freedom	7 per arm
Payload	0.5 kg
Reach	559 mm
Width, Length, height [mm]	$399 \times 449 \times 571$
Weight	38 kg
Max TCP Velocity	1.5 m/s
Max TCP Acceleration	11 m/s^2
Acceleration time 0-1 m/s	0.12 s
Position repeatability	0.02 mm

Table 6.1: Properties of YuMi.



(a) Workspace, front view.



(b) Workspace, side view.

Figure 6.2: YuMi's workspace. Figures from [71]

YuMi features an embedded robot controller, a FlexPedant control tablet and two Smart Grippers. The embedded controller is based on ABB's IRC5 robot controller and contains all the electronics required to control and drive the robot. The robot controller runs the RobotWare operating system. The robot controller and RobotWare comprise a closed system where the user has restricted access to the lower layers of the system. The embedded controller has two modes of operation: Manual mode and Automatic mode. In manual mode, the operator controls the robot via the FlexPendant. The operator can manually jog the robot's arm, manually grip and jog the grippers and perform other operations available through the FlexPendant. Programs loaded to the controller can be stepped through but not run normally. TCP speed is limited in manual mode to 250 mm/s [48]. Automatic mode allows programs to control the robot.

The FlexPendant is a hand-held control tablet that can control many of the functions involved with operating the robot system. The FlexPendant is connected to the robot controller via a wired connection and is equipped with a touch-screen, tactile buttons, an emergency stop button and a multi-directional joystick. The FlexPendant is a separate computer from the robot controller. The underlying OS of the FlexPendant is Windows CE, Microsoft's operating system for embedded devices [23]. A Graphical User Interface (GUI) runs on top of Windows CE and is the environment the user interacts with. The FlexPendant is particularly useful for starting and stopping program execution, debugging, monitoring of program execution, and jogging of the robot. Error messages are displayed on the screen in addition to log messages informing the user of events in the system. The FlexPendant can bee seen in Figure 6.3.

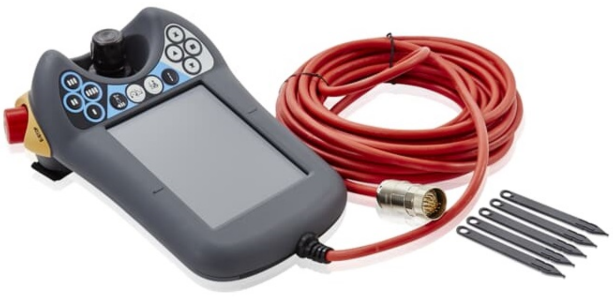
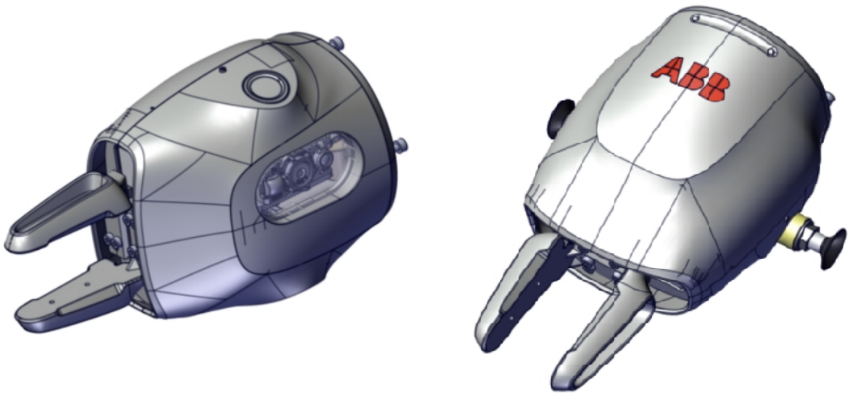


Figure 6.3: The FlexPendant.

The Smart Gripper is a multi-functional gripper specifically made for assembly and part handling. The gripper has a servo module and can optionally be equipped with a vision module and up to two vacuum modules. The vacuum modules are located on either side of the gripper and the vision module is located inside the palm. The maximum gripping force is 20 N, the combined maximum travel distance of the fingers is 50 mm and the maximum speed is 25 mm/s. The YuMi used in this work has two Smart Grippers. The right gripper is equipped with the vision module and the left with a top-mounted vacuum module. The Smart Gripper is depicted in Figure 6.1 and Figure 6.4.



(a) Side view. **(b)** Top view.

Figure 6.4: CAD drawing of a smart gripper equipped with the vision module and two suction modules.

Programming of ABB robots is done in the RAPID programming language, ABB's in-house developed programming language. RAPID programs consist of program modules and system modules. Program modules use the .mod extension and contain the executable code. System modules use the .sys extension and are used to define common data and routines, functioning similar to libraries. Program and system modules are organized into tasks. A task represents an application on the robot. Tasks can be linked to a mechanical unit group, which is a system parameter representing a mechanical device managed by the robot controller, typically an arm. While multiple tasks can be run in parallel, only one task is allowed motion control privileges for each mechanical unit group. RAPID programs can be written in the RobotStudio Integrated Development Environment (IDE). RobotStudio provides an environment for modeling, simulation, offline and online programming of ABB robots and robot cells. When programming offline, code can be tested on a simulated robot running on a virtual controller in a simulated environment.

6.1.2 Zivid One Structured Light 3D Camera

The Zivid One structured light 3D camera used in this work, facilitates rapid capture of accurate 3D point cloud data from a scene, as well as high resolution color images. The specified accuracy of the camera is 0.1 mm at a distance of 0.6 m, and the maximum acquisition frequency is 10 Hz. The camera has a wide array of settings which can be tuned in order to capture 3D data with low levels of noise under varying lighting conditions. In addition, the settings can be adjusted for difficult objects, such as very dark low texture objects, or shiny objects. This makes it suited for object detection and pose estimation tasks related to robotic bin picking in industrial settings.

The Zivid camera can be controlled using the Zivid SDK [7] which contains the core API used to implement camera functionality in our system. In addition to the API, the desktop application Zivid Studio is a useful tool for quick troubleshooting and calibration of the camera settings for optimal image acquisition.

Figure 6.5: Zivid One 3D camera

6.2 External Computer

The robot is controlled from an external desktop computer. The specifications are listed in Table 6.2.

Component	Type
RAM	16 GB DDR3 @ 1600 Mhz
CPU	Intel Core i7-4790K @ 4.00GHz
GPU	Nvidia GTX 780 Ti
OS	Ubuntu 18.04.4 LTS

Table 6.2: External computer specifications.

6.3 Software

6.3.1 External Control Interfaces of YuMi

ABB robots can be controlled through two external control interfaces, Robot Web Services (RWS) and Externally Guided Motion (EGM). RWS is a networked API that leverage the Hypertext Transfer Protocol (HTTP) to control aspects of a IRC5 robot controller [56]. The networked API is RESTful, meaning the API follow the Representational State Transfer (REST) specification. This places requirements on the resources to be directly accessible via an URI and for communication to be stateless [22]. A robot web service is defined by a resource URI and one of the REST-approved HTTP verbs defining the action to be performed on the resource. Responses are delivered in either the JSON or XML format. RWS enables control of aspects of the robot that normally would require physical access to the robot, for instance starting program execution, changing parameters, inspecting system info or stopping motors. RWS does not provide means for high frequency communication. The use of RWS require the robot controller to be in auto mode.

EGM is a low-latency, high-frequency motion control interface available for RobotWare through an additional licence [6]. When activated, RAPID instructions for setup and control will become available to the user, and a set of system settings will be installed. EGM offers two modes of operation, Position Guidance and Path Correction. Position Guidance provides a low-level interface to the robot controller, allowing robot motion to be controlled from an external device. Position guidance opens an interface directly to the motor reference generator, allowing an external device to issue motion references and control the robot in real-time. By issuing references directly to the low-level motion system, bypassing the internal path planner, reading and writing positions to the motion system can be done every 4 ms, with a control lag of 10-20 ms depending on the robot type. This is the lowest latency method for external motion control supported by ABB robot controllers. EGM Position Guidance can be used with joint references in joint mode and pose references in pose mode. Joint mode is supported by all ABB manipulators, while pose mode is only supported for 6-axis manipulators, and therefore unavailable to YuMi. Path Correction provides means to produce path adjustments to a preprogrammed path from data obtained from an external sensor. A corrected path is constructed from the correction data, and the robot is moved according to the adjusted path.

Using EGM Position Guidance, an external device transmits motion references to the robot controller and receives joint state feedback in the form of joint position and velocity, at a rate of 250 hz. Motion references and joint state feedback are transmitted by UDP transport in accordance to the EGM Sensor Protocol, an ABB developed application protocol designed for high-speed communication with minimal overhead. The protocol utilizes Google Protocol Buffers (Protobuf) for efficient serialization and deserialization of the messages. The structure of messages transmitted using protobuf are defined in proto definition files. The proto definition file is compiled by the protobuf compiler, generating code for serialization and deserialization of the message type. Both communication participants require this code. The Protobuf serialization procedure converts messages to a binary format. The EGM Sensor protocol features no built-in synchronization of requests and responses or handling of lost messages.

The external device and the robot controller operate in a server-client relationship. The external device launches an EGM server which must be available to the EGM client before a connection can be made. Via a RAPID command, the robot controller creates a client which connects to an EGM server located at a predefined IP address. After a connection has been established, the EGM client transmits information about the current joint states to the EGM server every 4 ms. The EGM server sends commands asynchronously to the client, which are executed by the motion control system. The data flow when using Position guidance is shown in Figure 6.6.

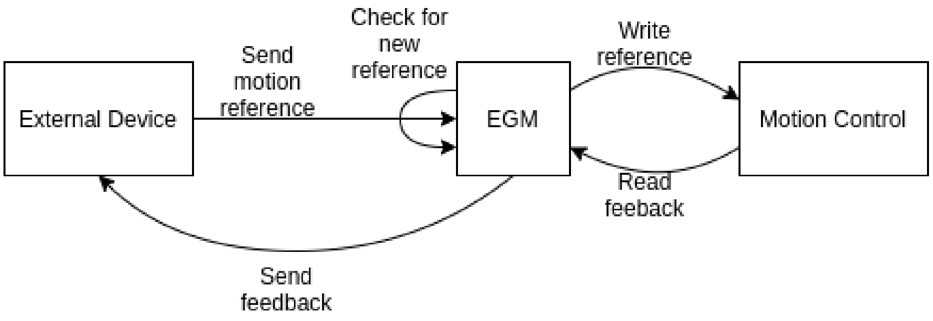


Figure 6.6: Data flow of EGM Position Guidance.

The EGM control system executes the desired motion by issuing joint speed commands to the motor control system. The control loop is based the following relation between speed and position [6, p. 343]

$$\text{speed} = \text{k} * (\text{ pos\_ref - pos }) + \text{ speed\_ref} \tag{6.1}$$

speed denotes the commanded speed sent to the motion control system from EGM. speed_ref denotes a reference speed fed forward in the control system. Likewise, pos_ref denotes a fed forward, reference position. pos denotes the desired position received from the external computer. The position gain factor k is composed of two configurable constants that are modifiable by the user.

$$\text{k} = \text{PCG} * \text{PPG} \tag{6.2}$$

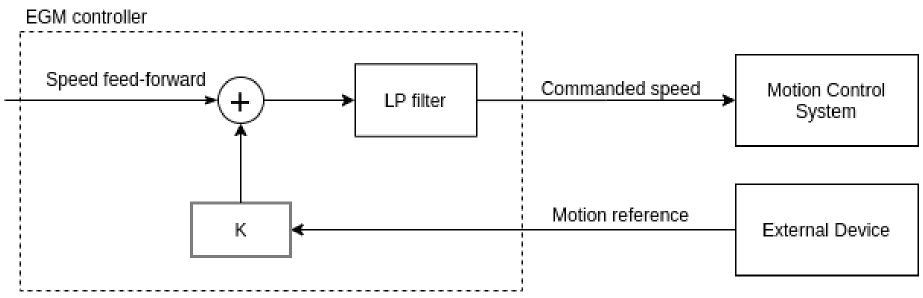


Figure 6.7: Simplified EGM control loop.

As shown in Equation (6.1), the Position Guidance control system controls the joint speed. However, both joint position and joint speed commands may be issued from the external device to induce motion via EGM Position Guidance. PPG denotes the proportional position gain constant specified in the system configuration of the robot controller, and cannot be altered at runtime. PCG denotes the Proportional Correction Gain RAPID parameter. PCG can be changed at runtime using RAPID instructions and can be used to tune the k-factor by adjusting how much of the proportional position gain is to be used. PCG can be set to 0 to control the joint velocities directly from the external device. Low pass filtering can be applied if the motion references are based on noisy sensor data. Using EGM Position Guidance, the responsibility of generating smooth trajectories are transferred to the user. The path planner is bypassed by EGM, meaning the manipulator's path is created directly from user input. The robot will react quickly to all position references sent to the controller, also faulty ones.

6.3.2 ROS 2 Robot Control Architecture

A ROS 2 robot control architecture to control YuMi from an external computer has been developed. The developed architecture is made up of tasks running on the robot controller, and a larger ROS 2 architecture on an external computer. Communication between YuMi and the external computer is done using EGM and RWS. The external computer and the robot are connected to the same LAN. Maintaining an open EGM connection requires that a RAPID program is continuously running an EGM client on the robot controller. To this end, the StateMachine RobotWare Add-In [62] has been deployed on the robot controller. The StateMachine Add-In provides a state machine implementation in RAPID code and is intended to be used with external devices using RWS, and optionally with EGM. The Add-In installs RAPID modules and system files, connects modules to tasks and mechanical unit groups, adds a set of I/O signals, and loads configurations to the system. A state machine can only manage one mechanical unit group, thus for YuMi, two instances of the state machine are loaded.

The state machine implemented in RAPID code is a finite-state machine. There are four possible states: "Initialize", "Idle", "Run EGM motion" and "Run RAPID routine". Transitions between the states are interrupt driven. Transitions between states can be invoked from the external computer, allowing the state machine to be externally controlled. A diagram showing the states and some state transitions is depicted in Figure 6.8.

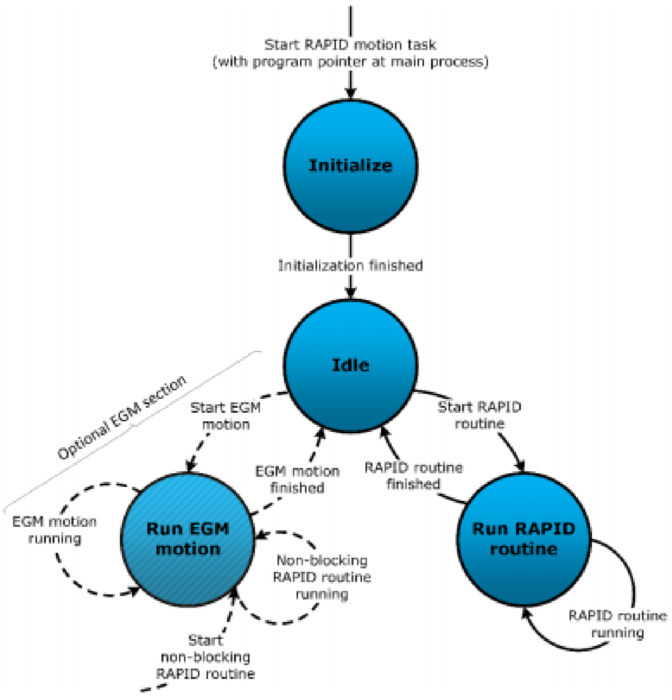


Figure 6.8: Diagram showing state machine sates and transitions. Figure from [66].

The state machine installed by the Add-In is made up of five RAPID modules: TRobMain, TRobEGM, TRobRAPID, TRobSG and TRobUtility. Additionally a optional watchdog module, TRobWatchdog is loaded. The modules are loaded into one task, which is then coupled to one mechanical unit group. The installation installs the modules needed by the system. If the robot controller does not have EGM enabled, or does not include smart grippers, their respective modules TRobEGM and TRobSG will not be installed. The modules are listed in Table 6.3.

Module	**Contains**
TRobMain	The program itself. Includes the "`main()`".
TRobEGM	Functions and variables related to EGM motion state.
TRobRAPID	Functions and variables related to RAPID routine state.
TRobSG	Functions and variables related to Smart Gripper operation.
TRobUtility	Commonly used functionalities and helper functions.
TRobWatchdog	Watchdog supervising the system.

Table 6.3: RAPID modules of the state machine.

In the "Initialize" state the modules of the state machine are initialized. The state is automatically entered as the main RAPID script is executed. After leaving this state, the state machine can never return to this state. The "Idle" state is the base operational mode entered when the state machine is successfully initialized. In this state, the TRobMain module executes an infinite loop. Available transitions from this state are "Start EGM motion" and "Start RAPID routine". The Idle state is the only state directly reachable from all states.

In the "Run RAPID Routine" state, the state machine can execute predefined RAPID routines loaded to the robot controller. Together with the I/O signal to trigger the transition to the state, the name of the rapid routine must be provided by the external device. Runtime-configurations for the RAPID routine can be set in a similar fashion. RAPID routines can be executed both in a blocking and asynchronous manner. When finished, the state machine returns to "Idle" automatically.

In the "Run EGM Motion" state, the state machine starts an EGM client and opens an EGM channel to the EGM server. Upon successful connection, received motion references are executed. If no commands are received between the time of connection and a defined time limit, the state machine will stop, and return to the "Idle" state. The "Run EGM Motion" state can be exited by an I/O signal triggering an EGM stop signal. As two state machines are needed for YuMi, two EGM clients are constructed on the robot controller each connecting to separate EGM servers.

Additional functionality is defined in "TRobSG" and "TRobUtility". "TRobSG" provides routines for operation of the smart grippers, such as calibration, jogging, gripping, and suction control. "TRobUtilities" contains commonly used utilities.

The ROS 2 robot control architecture is composed of seven modules: AbbEgmHardware, YumiRobotManager, SgControl, JointStateController, JointTrajectoryController, ParameterServer and GlobalJointStates. Owing to the need for two EGM servers, and to avoid left- and right-specific classes, the architecture is split into three ROS 2 namespaces. A namespace for the left arm, a namespace for the right and a global namespace for arm-agnostic modules. Both the left arm and right arm namespace contain an instance of AbbEgmHardware, SgControl, ParameterServer, JointStateController and JointTrajectoryController. The YumiRobotManager and GlobalJointStates modules exist in the global namespace. All modules in the left arm namespace "`/l`" are exclusively involved with the left arm and its resources, likewise in the right arm namespace "`/r`". The modules located in the global namespace are not associated with a specific arm. The separation of modules into the namespaces can bee seen in Figure 6.9.

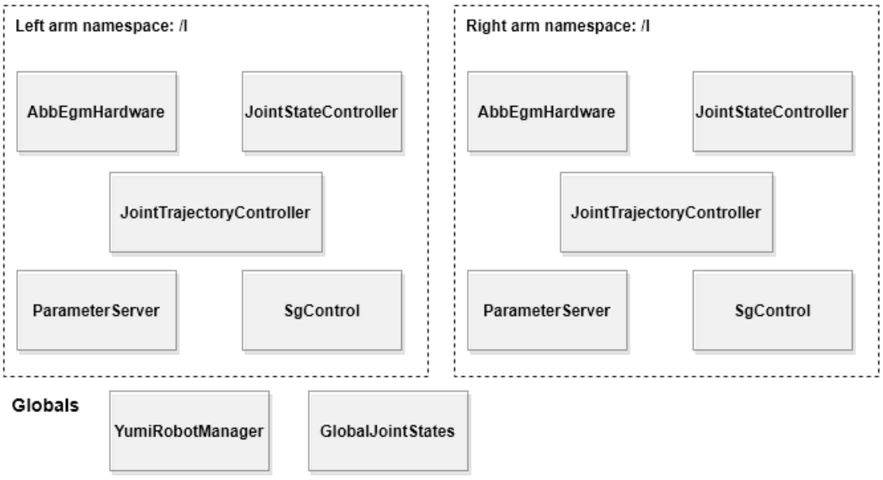
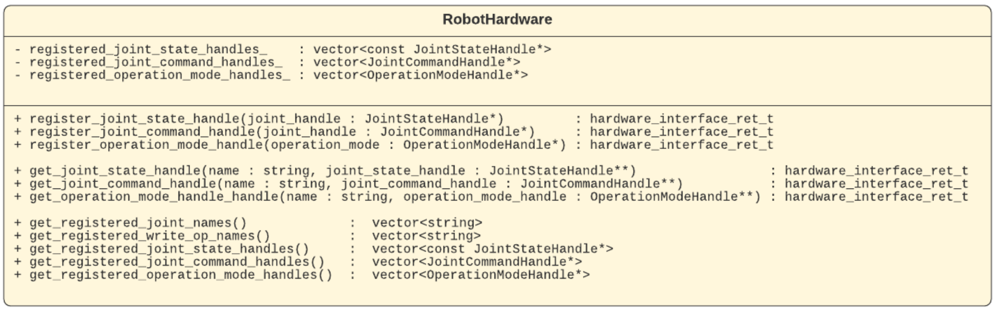


Figure 6.9: Figure showing the separation of modules into ROS 2 namespaces.

AbbEgmHardware

A central concept in ROS and ROS 2 is hardware abstraction. In the developed architecture, hardware abstraction is provided through the AbbEgmHardare class. AbbEgmHardware hosts a EGM server transmitting motion commands to an EGM client on YuMi. The component must be able to receive robot states and transmit motion commands in a low-latency and reliable manner. In order to achieve this, the component is built into the ROS 2 control framework, `ros2_control` [57]. `ros2_control` provides a framework for hardware abstraction, allows for implementation of robot-agnostic controllers, and provides capabilities to manage robot controllers with emphasis on real-time performance. Specifically, `ros2_control` is a collection of C++ ROS 2 packages containing robot-agnostic base classes with standardized interfaces. From these, robot-specific hardware interfaces can be derived and robot agnostic robot controllers can be integrated. Additionally, robot controllers can be developed once and implemented on different hardware.

The three main components in the framework are the two abstract classes RobotHardware and ControllerInterface, and the controller manager. Data is shared within the `ros2_control` framework using three custom "handle" data types: JointStateHandle, JointCommandHandle and OperationModeHandle. The handle data type facilitates efficient, copy-free data transfer. JointStateHandle contains pointers to the read data of a single joint, containing fields for joint position, joint velocity and joint torque. JointCommandHandle contains a pointer to the command to be issued to a joint. OperationModeHandle contains a pointer to the operation mode of the joint, allowing specification of read and write privileges to specific joints. The handles, and the data they point to, reside within the robot-specific hardware interface, which must be derived from the abstract RobotHardware class. RobotHardware provides a common interface towards all hardware interfaces. It contains pointers to the handles and methods for interacting with the handle-pointers. The methods form the interface which controllers are expecting to interact with, and are the key-enabler for robot-agnostic controllers. A UML diagram of the RobotHardware class can be seen in Figure 6.10.


```
RobotHardware

- registered_joint_state_handles_     : vector<const JointStateHandle*>
- registered_joint_command_handles_   : vector<JointCommandHandle*>
- registered_operation_mode_handles_  : vector<OperationModeHandle*>

+ register_joint_state_handle(joint_handle : JointStateHandle*)          : hardware_interface_ret_t
+ register_joint_command_handle(joint_handle : JointCommandHandle*)      : hardware_interface_ret_t
+ register_operation_mode_handle(operation_mode : OperationModeHandle*) : hardware_interface_ret_t

+ get_joint_state_handle(name : string, joint_state_handle : JointStateHandle**)                : hardware_interface_ret_t
+ get_joint_command_handle(name : string, joint_command_handle : JointCommandHandle**)          : hardware_interface_ret_t
+ get_operation_mode_handle_handle(name : string, operation_mode_handle : OperationModeHandle**) : hardware_interface_ret_t

+ get_registered_joint_names()              :  vector<string>
+ get_registered_write_op_names()           :  vector<string>
+ get_registered_joint_state_handles()      :  vector<const JointStateHandle*>
+ get_registered_joint_command_handles()    :  vector<JointCommandHandle*>
+ get_registered_operation_mode_handles()   :  vector<OperationModeHandle*>
```

Figure 6.10: UML class diagram of the abstract RobotHardware class.

The derived robot specific hardware interface contains hardware-specific methods for communicating with the hardware, wrapped in a simple `init()`, `read()` and `write()` API. AbbEgmHardware is the developed robot-specific hardware interface. It is designed to be the interface to a single arm, meaning two instances are required to control both arms. The EGM communication is implemented using the open-source `abb_libegm` [2] C++ library, developed and maintained by ABB. It provides all the necessary resources to set up an EGM server and communicate with an EGM client. AbbEgmHardware uses the high-level API EGMControllerInterface available through `abb_libegm`. EGMControllerInterface provides methods for launching an EGM server, establishing connection with an EGM client, reading of robot state and transmission of commands.

The class is arm-agnostic. It represents a specific arm by loading arm-specific parameters from the namespaced ParameterServer. The ParameterServer loads a configuration file containing the arm-specific parameters and offers the parameters through services. Interaction with the ParameterServer is only performed at startup, in the `init()` method. After the required information about the arm is retrieved from the ParameterServer, the method constructs the EGM server and handles, registers pointers to these in RobotHardware, and awaits a connection from an EGM client. The `init()` function call is blocking until a successful connection between the EGM client and the EGM server has been made.

AbbEgmHardware's `read()` method waits until a message from the EGM client is received or a timeout limit is reached. If a message is received within the timeout limit, using EGMControllerInterface's `read()` method, the state is read into a `abb_libegm` input-container residing in the class. The contents are then deserialized in accordance with the EGM Sensor Protocol and written into the registered joint state handles. AbbEgmHardware's `write()` method copies the contents of the joint command handles to a `abb_libegm` output-container, whose contents are serialized in accordance to the EGM Sensor Protocol and sent to the EGM client using EGMControllerInterface's `write()` method. A simplified UML diagram of AbbEgmHardware is depicted in Figure 6.11.

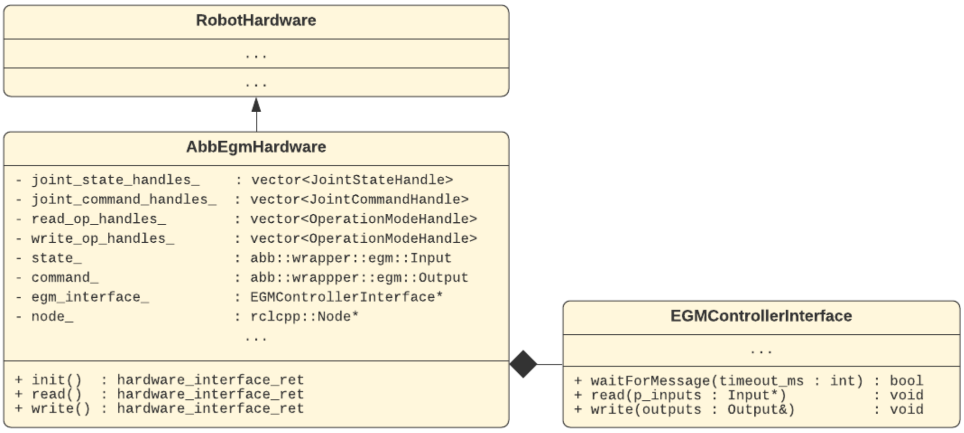

RobotHardware
...
...
AbbEgmHardware
- joint_state_handles_ : vector<JointStateHandle>
- joint_command_handles_ : vector<JointCommandHandle>
- read_op_handles_ : vector<OperationModeHandle>
- write_op_handles_ : vector<OperationModeHandle>
- state_ : abb::wrapper::egm::Input
- command_ : abb::wrappper::egm::Output
- egm_interface_ : EGMControllerInterface*
- node_ : rclcpp::Node*
...
+ init() : hardware_interface_ret
+ read() : hardware_interface_ret
+ write() : hardware_interface_ret
EGMControllerInterface
...
+ waitForMessage(timeout_ms : int) : bool
+ read(p_inputs : Input*) : void
+ write(outputs : Output&) : void

Figure 6.11: Simplified UML diagram of the AbbEgmHardware class.

Controllers used within the `ros2_control` framework must inherit from the ControllerInterface class. ControllerInterface provides a standardized interface towards controllers. The ControllerInterface class is derived from the LifecycleNodeInterface base class, allowing it's lifecycle to be managed similar to a LifecycleNode. Setup and initialization is split into `on_configure()` and `on_activate()` methods. By having all controllers in this format, a single entity can manage all controllers simultaneously. This is handled by the ControllerManager. The ControllerManager is given a pointer to the hardware interface in it's constructor. After construction, the ControllerManager has four responsibilities: loading, configuring, and activating the controllers, and applying the `update()` method in the control loop. The `update()` method applies the controllers' update law and computes the commands to be issued. The ControllerManager calls `update()` for all activated controllers. Thus it enables the use multiple controllers in conjunction. The ControllerManager can at any moment deactivate or activate controllers, allowing controllers to be hot-swapped during runtime.

AbbEgmHardware, the ControllerManager and the controllers are run in the same executable. The executable runs a control loop at 250 hz. In each iteration, the current robot joint states are read and joint commands calculated and transmitted to the EGM client. A simplified version of this executable can be seen in Listing 6.1.

```
int main(){

  robot = AbbEgmHardware();
  robot.init();

  ControllerManager controller_manager(&AbbEgmHardware);

  **Controllers are loaded, configured and activated**

  // The control loop.
  while (rclcpp::ok()){
    robot.read();
    controller_manager.update();
    robot.write();
  }

  **Teardown.**
  return 0;
}
```

Listing 6.1: Simplified implementation of the control loop executable.

JointStateController and JointTrajectoryController

Two controllers are implemented. The purpose of the JointStateController is to publish read joint states. In it's `update()` method, it reads the JointStateHandles and generates a JointState message which is published on a namespaced `/joint_states topic`.

JointTrajectoryController provides the commands to be issued to the robot. The controller subscribes to a `/joint_trajectory_controller/joint_trajectory` topic within its namespace. On this topic it receives JointTrajectory messages containing a vector of the joint names and a vector of waypoints specifying joint position, joint velocity and joint acceleration of the joints, at specific timestamps.

When it's `update()` method is called, the controller finds the next waypoint in the trajectory. When a new waypoint in the trajectory is reached, the JointCommandHandle of the hardware interface is updated with the joint position specified in the waypoint. The update law of the JointTrajectoryController can thus be said to be a timestamp-managed joint position control. In addition, two more features are introduced. The first feature is the ability for a external ROS 2 node to stop the execution of a trajectory and to remove it from the controller. The feature is topic-activated, where a subscriber in the controller listens for a Bool message on a `joint_trajectory_controller/arm_stop` topic within the namespace. If a message containing "true" is received, the execution of the trajectory is stopped. A data flow diagram of the control loop for the right arm, showing the controllers and their interactions with the hardware interface and their topics is depicted in Figure 6.12.

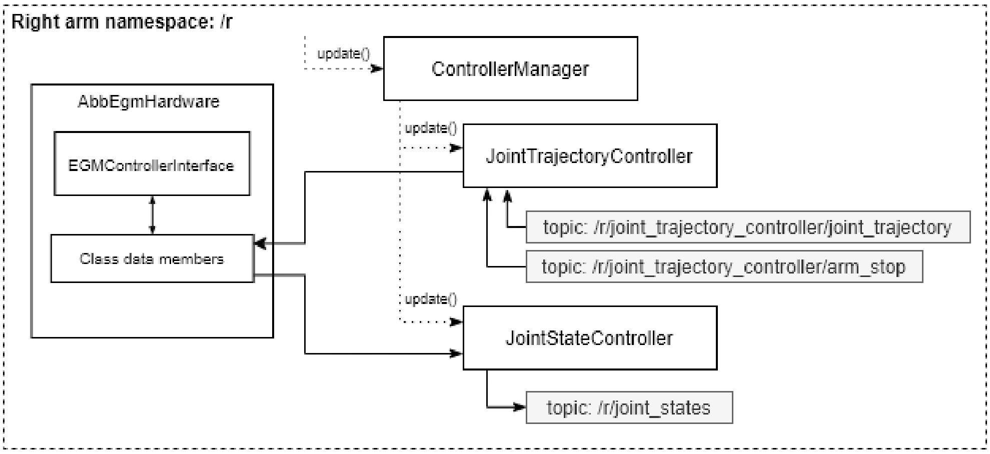
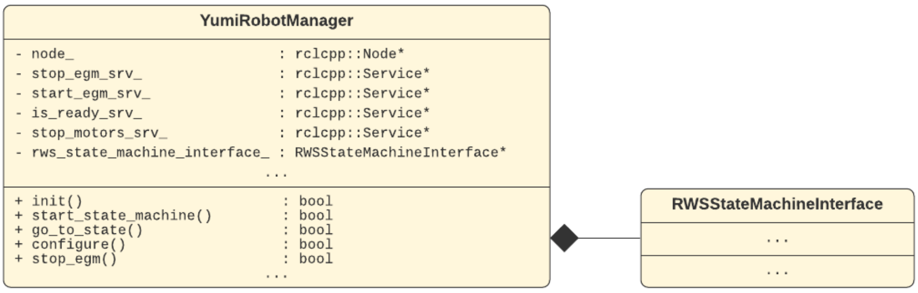


Figure 6.12: Data flow diagram of the complete `ros2_control` control loop.

YumiRobotManager and SgControl

YumiRobotManager is a module providing general robot-management functionalities. SgControl is used to control a Smart Gripper. Both YumiRobotManager and SgControl communicates with the robot controller through its RWS interface. The RWS communication is implemented using the open-source `abb_librws` C++ library [3]. The library is developed and maintained by ABB, and contains implementations facilitating use of RWS. `abb_librws` contains two high-level API classes: RWSInterface and RWSStateMachineInterface. RWSInterface provides most of the RWS functionality through its methods. The methods abstract away the details of the HTTP-based communication wrapping them in convenient function calls. RWSStateMachineInterface inherits from RWSInterface. It provides methods for interaction with a state machine. It is aware of the state machine specific RAPID variables, routines and system configurations, and offers state machine specific services like invoking state transitions.

During initialization the YumiRobotManager ensures that both state machines are in the "Idle" state. Issuing of the "Start EGM Motion" state machine transition signal is offered as a ROS 2 service. Furthermore, ROS 2 services for transitioning the state machine out of the "Run EGM motion" state and for checking if state machine is currently in the "Idle" state is provided. A simplified UML class diagram can bee seen in Figure 6.13.



YumiRobotManager
- node_ : rclcpp::Node*
- stop_egm_srv_ : rclcpp::Service*
- start_egm_srv_ : rclcpp::Service*
- is_ready_srv_ : rclcpp::Service*
- stop_motors_srv_ : rclcpp::Service*
- rws_state_machine_interface_ : RWSStateMachineInterface*
...
+ init() : bool
+ start_state_machine() : bool
+ go_to_state() : bool
+ configure() : bool
+ stop_egm() : bool
...
RWSStateMachineInterface
...
...

Figure 6.13: Simplified UML diagram of the YumiRobotManager class.

SgControl is a class featuring methods for controlling a smart gripper. The control of the Smart Gripper is offered to the rest of the ROS 2 system through an action and topic interface. Gripping is offered with a grip action. Jogging of the gripper is offered through a topic interface. SgControl subscribes to a `/jog_gripper` topic within the namespace. The topic contains information about the desired position of the gripper. The current gripper position is published to a `/gripper_pos` topic within the namespace. SgControl controls the grippers using the RWSStateMachineInterface. A simplified UML class diagram of SgControl can bee seen in Figure 6.14.

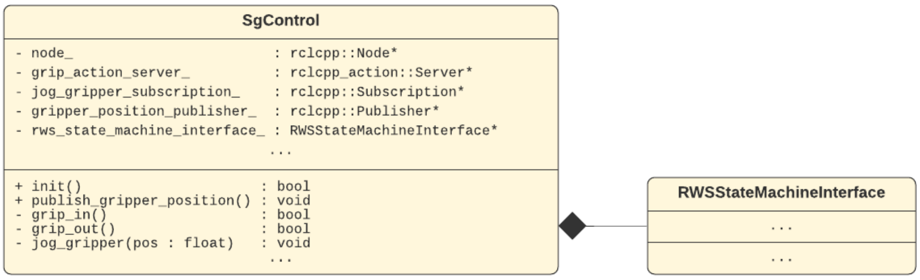

SgControl
- node_ : rclcpp::Node*
- grip_action_server_ : rclcpp_action::Server*
- jog_gripper_subscription_ : rclcpp::Subscription*
- gripper_position_publisher_ : rclcpp::Publisher*
- rws_state_machine_interface_ : RWSStateMachineInterface*
...
+ init() : bool
+ publish_gripper_position() : void
- grip_in() : bool
- grip_out() : bool
- jog_gripper(pos : float) : void
...
RWSStateMachineInterface
...
...

Figure 6.14: Simplified UML diagram of the SgControl class.

The user-level API for utilizing the YumiRobotManager and SgControl modules within ROS 2 is provided by two classes, RobotManagerClient and GripperClient. The classes offer methods to use the functionalities provided by YumiRobotManager and SgControl respectively. A simplified UML diagram of RobotManagerClient and GripperCient is depicted in Figure 6.15.

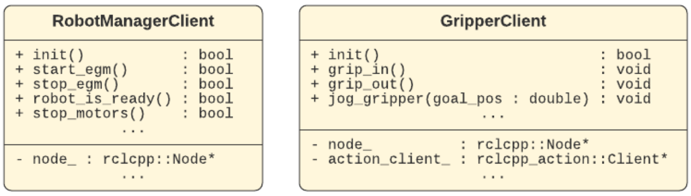
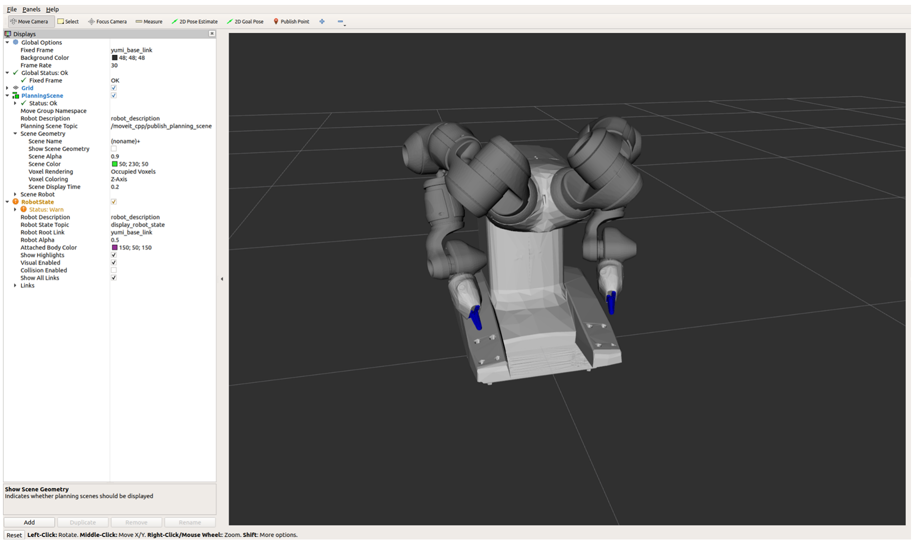


Figure 6.15: Simplified UML diagrams of the RobotManagerClient and GripperClient classes.

Simulated Backend and Visualization

Offline robot programming allows new algorithms and applications to be tested safely in a simulated environment. Offline robot programming can significantly reduce the downtime caused by reconfiguration and reprogramming of industrial robots. In the developed ROS robot control architecture, this is achieved by robot visualization using Rviz2 [58] and a simulated backend. Rviz2 is commonly used for robot visualization in ROS 2. Rviz2 loads a Universal Robot Description Format (URDF) file detailing the kinematics and inertial properties of the robot and STL 3D model representations the robot links. URDF is an XML-variant used extensively in the ROS and ROS 2 ecosystems. Rviz2 can visualize the robot state by subscribing to a topic reporting the joint state messages, but can also visualize point clouds from 3D cameras and other sensor data.

In the developed ROS 2 robot control architecture, rviz2 is configured to visualize YuMi by listening on a global `/joint_states` topic. The topic is created by the GlobalJointStates module, which subscribers to left and right arm joint states and finger positions, and publishes a combined JointState message on the global `/joint_states` topic. A screenshot of YuMi visualized in rviz2 can bee seen in fig. 6.16.



Global Options	
Fixed Frame	yumi_base_link
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
PlanningScene	✓
✓ Status: Ok	
Move Group Namespace	
Robot Description	robot_description
Planning Scene Topic	/moveit_cpp/publish_planning_scene
Scene Geometry	
Scene Name	(noname)+
Show Scene Geometry	☐
Scene Alpha	0.9
Scene Color	🟩 50; 230; 50
Voxel Rendering	Occupied Voxels
Voxel Coloring	Z-Axis
Scene Display Time	0.2
Scene Robot	
RobotState	✓
Status: Warn	
Robot Description	robot_description
Robot State Topic	display_robot_state
Robot Root Link	yumi_base_link
Robot Alpha	0.5
Attached Body Color	🟪 150; 50; 150
Show Highlights	✓
Visual Enabled	✓
Collision Enabled	☐
Show All Links	✓
Links	

Show Scene Geometry
Indicates whether planning scenes should be displayed

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift**: More options.

Figure 6.16: Screenshot of YuMi visualized in Rviz2.

A simulated backend has been developed to facilitate offline programming. The backend is organized in the same manner as the architecture for controlling the physical robot. When simulated, AbbEgmHardware writes its commands directly into the read-state container. YumiRobotManager offer the same services simulating the hardware response. SgControl simulate gripping and jogging functionalities. For the ROS 2 application, using the interfaces, the simulated backend cannot be differentiated from the normal architecture.

6.3.3 Motion Planning System

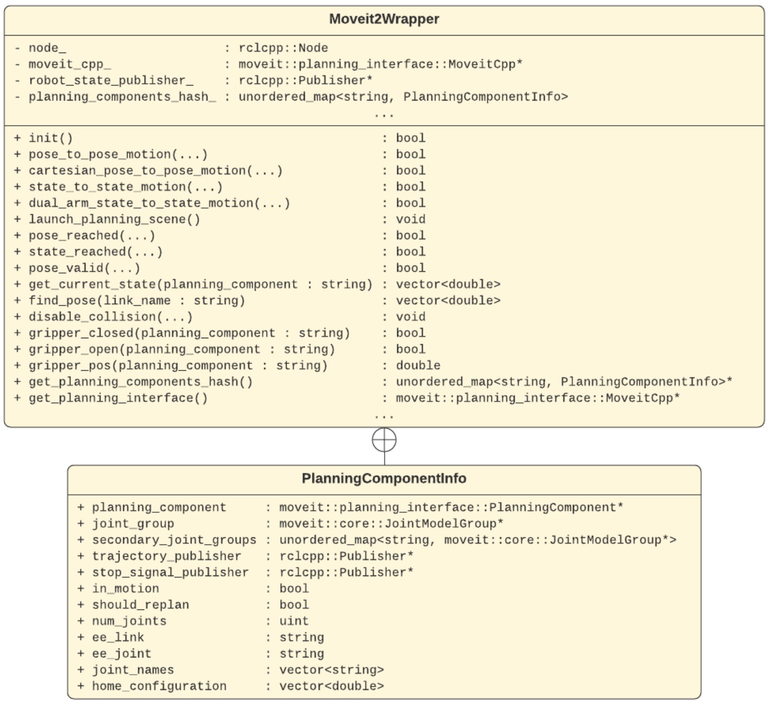
This section will present the developed motion planning system built using MoveIt 2 Beta release. The motion planning system features an implementation of the OMPL-based MoveIt 2 motion planning pipeline outlined in Section 4.5.1 and the linear Cartesian MoveIt 2 motion planning pipeline outlined in Section 4.5.2. The motion planning system is integrated with developed ROS 2 robot control architecture presented in Section 6.3.2. Together these provide a unified robot control user interface.

The motion planning system is comprised of three classes: Moveit2Wrapper, ObjectManager and MotionCoordinator. Moveit2Wrapper is the robot-centric MoveIt 2 interface, containing the implementations of the OMPL-based and linear Cartesian MoveIt 2 motion planning pipelines. ObjectManager manages objects in the planning scene and offers methods to manipulate these using MoveIt 2 interfaces. MotionCoordinator provides a high-level user API combining the functionalities of Moveit2Wrapper and ObjectManager to provide more complex task specific functionality. This API is used to develop applications utilizing the ROS 2 robot control architecture.

Moveit2Wrapper

Moveit2Wrapper wraps the `moveit_cpp` API and other interfaces of MoveiIt 2 in order to a create robot-centric API towards the MoveIt 2 motion planning framework. It provides capabilities to plan and command motions, poll information about the robot state and manage collision avoidance for the robot. It is initialized through a `init()` function call, during which a planning scene containing the robot's static environment is launched. The modelled environment makes the system aware of static obstacles within the robot workspace.

Moveit2Wrapper can be configured to be used with any robot prepared for control through the MoveIt 2 framework. Planning and motion are initiated and commanded using the PlanningComponent component of MoveIt 2 and an associated JointTrajectoryPublisher. A PlanningComponent is a planning unit representing a joint group, for instance an arm, and the JointTrajectoryPublisher is responsible for publishing the found solution. The published solution is executed by the JointTrajectoryController. PlanningComponentInfo is a structure defined in Moveit2Wrapper used to encapsulate a PlanningComponent and the associated JointTrajectoryPublisher. Additionally, to further streamline the interactions with the MoveIt 2 framework, other PlanningComponent information is also included. All PlanningComponents to be controlled are registered in a hash table using the associated name as key. A simplified UML class diagram of Moveit2Wrapper and PlanningComponentInfo can be seen in Figure 6.17.



```
Moveit2Wrapper
- node_                      : rclcpp::Node
- moveit_cpp_                : moveit::planning_interface::MoveitCpp*
- robot_state_publisher_     : rclcpp::Publisher*
- planning_components_hash_  : unordered_map<string, PlanningComponentInfo>
                                    ...
+ init()                                         : bool
+ pose_to_pose_motion(...)                       : bool
+ cartesian_pose_to_pose_motion(...)             : bool
+ state_to_state_motion(...)                     : bool
+ dual_arm_state_to_state_motion(...)            : bool
+ launch_planning_scene()                        : void
+ pose_reached(...)                              : bool
+ state_reached(...)                             : bool
+ pose_valid(...)                                : bool
+ get_current_state(planning_component : string) : vector<double>
+ find_pose(link_name : string)                  : vector<double>
+ disable_collision(...)                         : void
+ gripper_closed(planning_component : string)    : bool
+ gripper_open(planning_component : string)      : bool
+ gripper_pos(planning_component : string)       : double
+ get_planning_components_hash()                 : unordered_map<string, PlanningComponentInfo>*
+ get_planning_interface()                       : moveit::planning_interface::MoveitCpp*
                                    ...

PlanningComponentInfo
+ planning_component     : moveit::planning_interface::PlanningComponent*
+ joint_group            : moveit::core::JointModelGroup*
+ secondary_joint_groups : unordered_map<string, moveit::core::JointModelGroup*>
+ trajectory_publisher   : rclcpp::Publisher*
+ stop_signal_publisher  : rclcpp::Publisher*
+ in_motion              : bool
+ should_replan          : bool
+ num_joints             : uint
+ ee_link                : string
+ ee_joint               : string
+ joint_names            : vector<string>
+ home_configuration     : vector<double>
```

Figure 6.17: Simplified UML diagram of the Moveit2Wrapper class.

The most important functionalities offered by Moveit2Wrapper are the motion methods implementing the OMPL-based and linear Cartesian MoveIt 2 motion planning pipelines outlined in Section 4.5.1 and Section 4.5.2. Moveit2Wrapper offers three motion methods applying the OMPL-based pipeline to plan and execute motion: `state_to_state_motion()`, `pose_to_pose_motion()` and the YuMi-specific `dual_arm_state_to_state_motion()`. `state_to_state_motion()` and `pose_to_pose_motion()` differ only on how the goal is defined, and implements the same OMPL-based motion planning pipeline. Initially, the goal is defined using a Constraint message, where either a state or a pose is imposed as the constraint for the goal. Next, it attempts to plan a trajectory to the constrained goal. If unsuccessful a set number of retries are allowed. If desired, and no other PlanningComponent is in motion, the planned motion can be visualized before execution. Finally the found trajectory is published using the JointTrajectory-Publisher associated with the PlanningComponent. The planned motion can be executed asynchronously or in a blocking manner. When called in a blocking manner the methods will not return until the commanded motion is complete. When called asynchronously the methods returns after the trajectory is published. While specifically written for the ABB YuMi, the dual-arm method follow the same steps. The execution flow is illustrated in Figure 6.18.

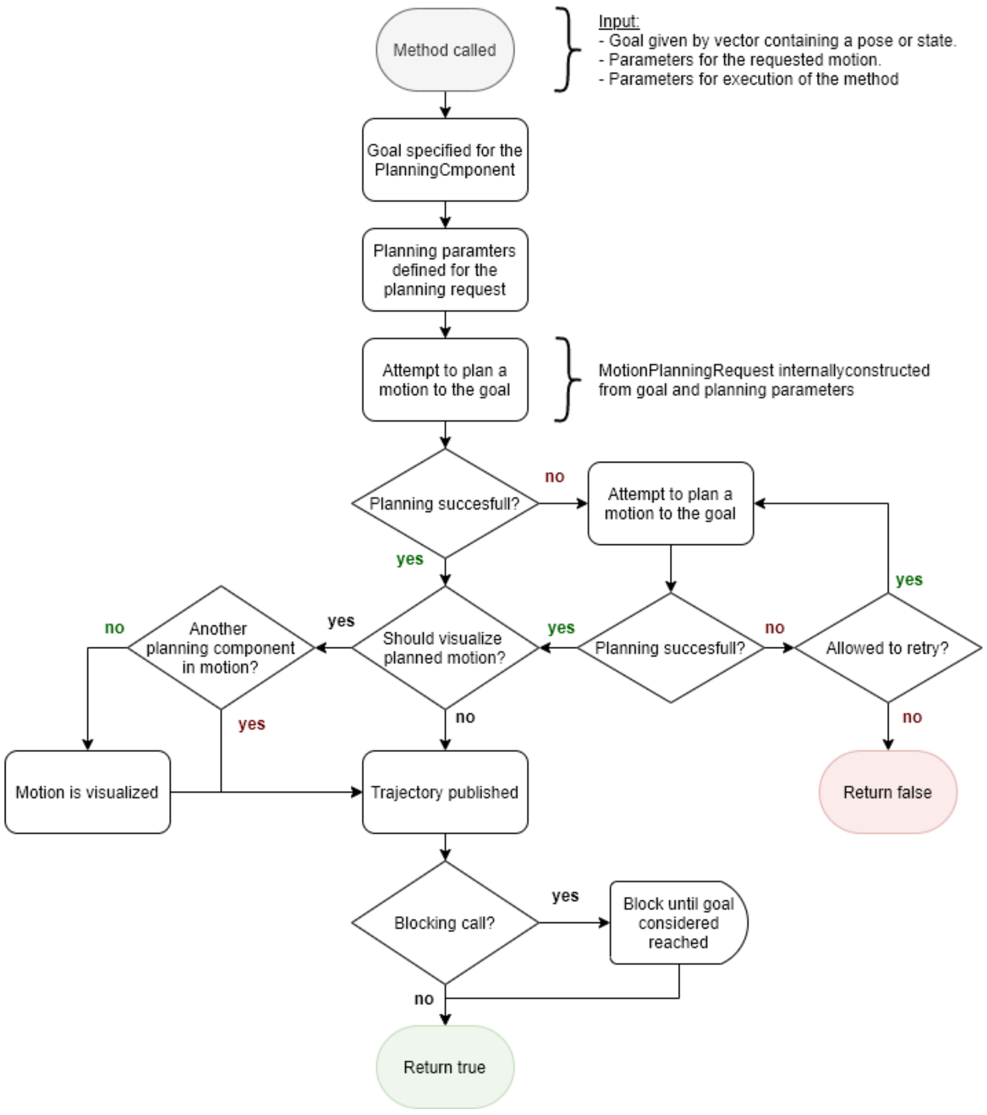

Method called
Input:
- Goal given by vector containing a pose or state.
- Parameters for the requested motion.
- Parameters for execution of the method
Goal specified for the PlanningCmponent
Planning paramters defined for the planning request
Attempt to plan a motion to the goal
MotionPlanningRequest internallyconstructed from goal and planning parameters
Planning succesfull?
no
yes
Attempt to plan a motion to the goal
Planning succesfull?
no
yes
Allowed to retry?
yes
no
Return false
Should visualize planned motion?
yes
no
Another planning component in motion?
no
yes
Motion is visualized
Trajectory published
Blocking call?
yes
Block until goal considered reached
no
Return true

Figure 6.18: Flow chart of the execution flow of an OMPL-based motion method.

The linear Cartesian motion planning pipeline is implemented in the `cartesian_pose_to_pose_motion()` method. It takes a goal pose as input together with parameters defining the maximum allowed step size between waypoints in the the Cartesian path, and the maximum allowed step size in the configuration space during computation. When a path has been interpolated, a series of checks are applied. Initially, a check is performed to verify that the found path reaches the goal pose. If passed, the linearity of the path is assessed. If sufficiently linear, the path is collision checked. If deemed collision-free, the path is time-parameterized using the addIterativeSplineParameterization planning adapter. If desired, and no other planning component is in motion, the planned motion can be visualized before execution. Finally, the found trajectory is published using the JointTrajectoryPublisher associated with the PlanningComponent. Similar to the OMPL-based motion methods, the planned motion can be executed asynchronously or in a blocking manner. The execution flow is illustrated in Figure 6.19.

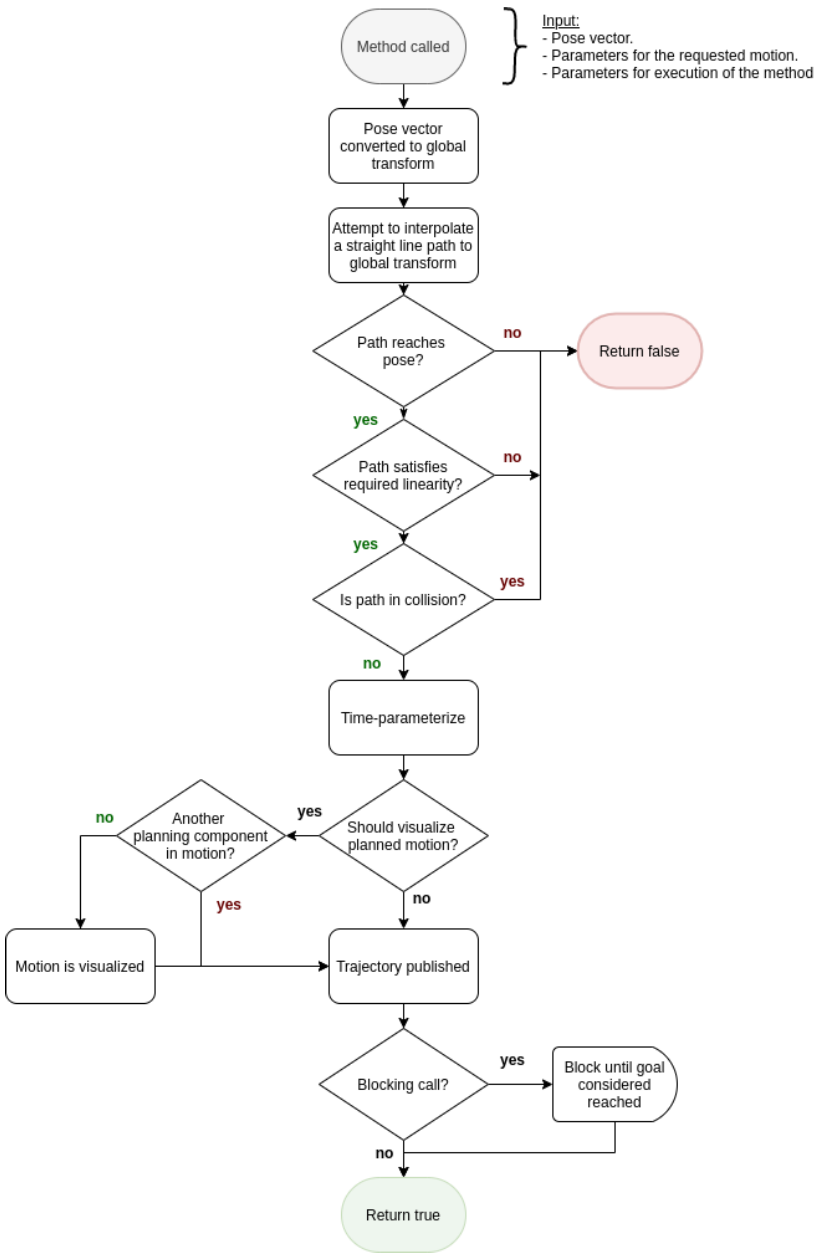

Method called
Input:
- Pose vector.
- Parameters for the requested motion.
- Parameters for execution of the method
Pose vector converted to global transform
Attempt to interpolate a straight line path to global transform
Path reaches pose?
no
Return false
yes
Path satisfies required linearity?
no
yes
Is path in collision?
yes
no
Time-parameterize
Should visualize planned motion?
yes
Another planning component in motion?
no
yes
no
Motion is visualized
Trajectory published
Blocking call?
yes
Block until goal considered reached
no
Return true

Figure 6.19: Flow chart of the execution flow of a linear Cartesian motion method.

ObjectManager

ObjectManager is the object-centric interface to the MoveIt 2 framework. It provides capabilities to load and add 3D models as meshes to the planning scene, keeps an updated registry of the objects in the scene and offers methods to manage the objects and modify their properties. The object registry consists of three hash tables: a hash table of registered meshes and their info, a hash table of registered solid primitives and their info and an object info lookup table. The mesh register uses a string containing the identifier of the object as key, and stores a MeshData data type containing the identifier and the extents of the mesh. The solid primitive register likewise uses a string containing the identifier of the object as key, and stores a SolidPrimitiveData data type containing the identifier, the dimensions of the solid primitive and the type of solid primitive. The object info lookup table uses a string containing the identifier of the object as key, and stores a ObjectData data type. An ObjectData contains the identifier, a bool indicating if its a collision object, the object type, the last observed pose of the object.

When initialized, the class loads all STL 3D models found in the companion ROS 2 package "`object_files`" as meshes. The meshes are registered appropriately in the registers, but not yet added to the planning scene. Adding objects to the planning scene, removal of objects from the planning scene, displacement and other manipulation of a registered object can be done through the API at runtime. A simplified UML class diagram of ObjectManager is depicted in Figure 6.20.

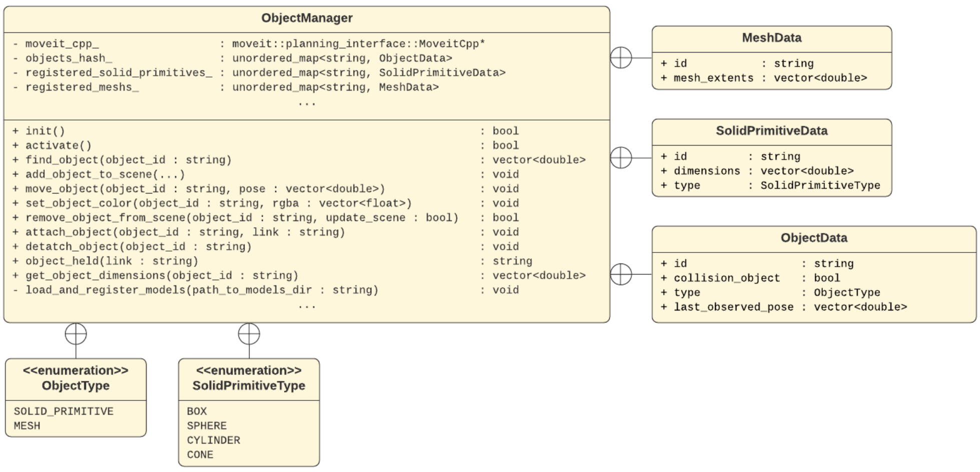

ObjectManager
- moveit_cpp_ : moveit::planning_interface::MoveitCpp*
- objects_hash_ : unordered_map<string, ObjectData>
- registered_solid_primitives_ : unordered_map<string, SolidPrimitiveData>
- registered_meshs_ : unordered_map<string, MeshData>
...
+ init() : bool
+ activate() : bool
+ find_object(object_id : string) : vector<double>
+ add_object_to_scene(...) : void
+ move_object(object_id : string, pose : vector<double>) : void
+ set_object_color(object_id : string, rgba : vector<float>) : void
+ remove_object_from_scene(object_id : string, update_scene : bool) : bool
+ attach_object(object_id : string, link : string) : void
+ detatch_object(object_id : string) : void
+ object_held(link : string) : string
+ get_object_dimensions(object_id : string) : vector<double>
- load_and_register_models(path_to_models_dir : string) : void
...
MeshData
+ id : string
+ mesh_extents : vector<double>
SolidPrimitiveData
+ id : string
+ dimensions : vector<double>
+ type : SolidPrimitiveType
ObjectData
+ id : string
+ collision_object : bool
+ type : ObjectType
+ last_observed_pose : vector<double>
<<enumeration>> ObjectType
SOLID_PRIMITIVE
MESH
<<enumeration>> SolidPrimitiveType
BOX
SPHERE
CYLINDER
CONE

Figure 6.20: Simplified UML diagram of the ObjectManager class.

MotionCoordinator

MotionCoordinator encompasses several modules reviewed so far. It contains an instance of the RobotManagerClient, two instances of the GripperClient, an instance of the KdlWrapper (will be presented in section 6.3.5), an instance of the Moveit2Wrapper and an instance of the ObjectManager. MotionCoordinator unifies robot control and motion planning in a high level API. This API is used to access all robot system functionality required for task specific application development. A simplified UML class diagram of MotionCoordinator is depicted Figure 6.21.

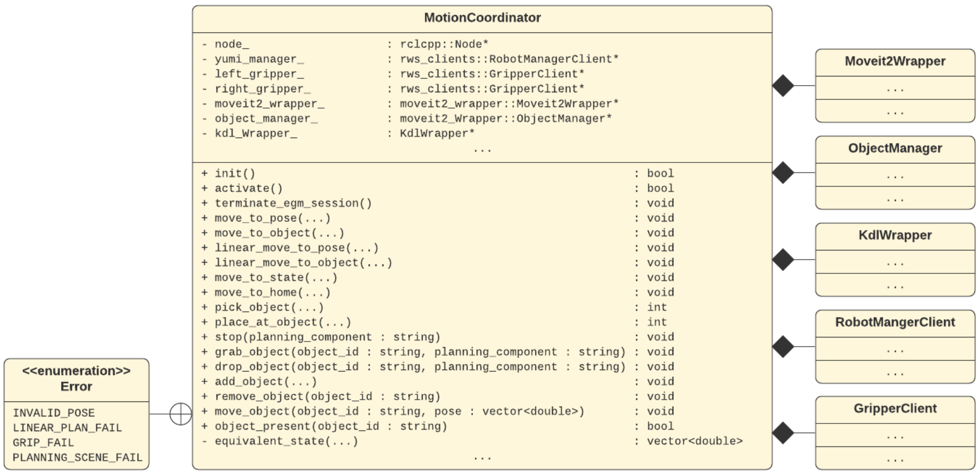

MotionCoordinator
- node_ : rclcpp::Node*
- yumi_manager_ : rws_clients::RobotManagerClient*
- left_gripper_ : rws_clients::GripperClient*
- right_gripper_ : rws_clients::GripperClient*
- moveit2_wrapper_ : moveit2_wrapper::Moveit2Wrapper*
- object_manager_ : moveit2_Wrapper::ObjectManager*
- kdl_Wrapper_ : KdlWrapper*
...
+ init() : bool
+ activate() : bool
+ terminate_egm_session() : void
+ move_to_pose(...) : void
+ move_to_object(...) : void
+ linear_move_to_pose(...) : void
+ linear_move_to_object(...) : void
+ move_to_state(...) : void
+ move_to_home(...) : void
+ pick_object(...) : int
+ place_at_object(...) : int
+ stop(planning_component : string) : void
+ grab_object(object_id : string, planning_component : string) : void
+ drop_object(object_id : string, planning_component : string) : void
+ add_object(...) : void
+ remove_object(object_id : string) : void
+ move_object(object_id : string, pose : vector<double>) : void
+ object_present(object_id : string) : bool
- equivalent_state(...) : vector<double>
...
Moveit2Wrapper
ObjectManager
KdlWrapper
RobotMangerClient
GripperClient
<<enumeration>>
Error
INVALID_POSE
LINEAR_PLAN_FAIL
GRIP_FAIL
PLANNING_SCENE_FAIL

Figure 6.21: Simplified UML diagram of the MotionCoordinator class.

During initialization of the MotionCoordinator instance, all interfaces and system modules are initialized. During activation, the StartEgm service is called, the planning scene containing the static robot environment is launched and the ObjectManager instance activated.

MotionCoordinator utilize functionality offered in Moveit2Wrapper to plan and execute motion. A planning goal can be defined as pose, a state or an object registered in the ObjectManager. Different methods are invoked depending on the goal. If the goal is defined as a pose either the `move_to_pose()` or `linear_move_to_pose()` method is used depending on the path requirements. If the goal is defined as a state, the `move_to_state()` method is used. If the goal is defined as an object, either the `move_to_object()` or `linear_move_to_object()` methods are used depending on the path requirements, to move to the object's pose.

For nonlinear pose-to-pose and state-to-state motions, an initial test checking if the registered end-effector of the planning component is already at the goal is performed. When the goal is specified using a pose or a state, this is done directly. When the goal is specified using a registered object, the planning scene is polled for the object, and the registered end-effector of the planning component is checked against a pose with a user-specified addition in the positive z-direction over the object pose. If passed robot motion is initiated. Event-triggered replanning can be activated for the motion. The event triggering replanning can for instance be a change in the planning scene[1]. Replanning allows for a path to be altered during motion execution. The execution flow is illustrated in Figure 6.22. A blue star indicates where the pose of the object is polled when the goal is given as a registered object.

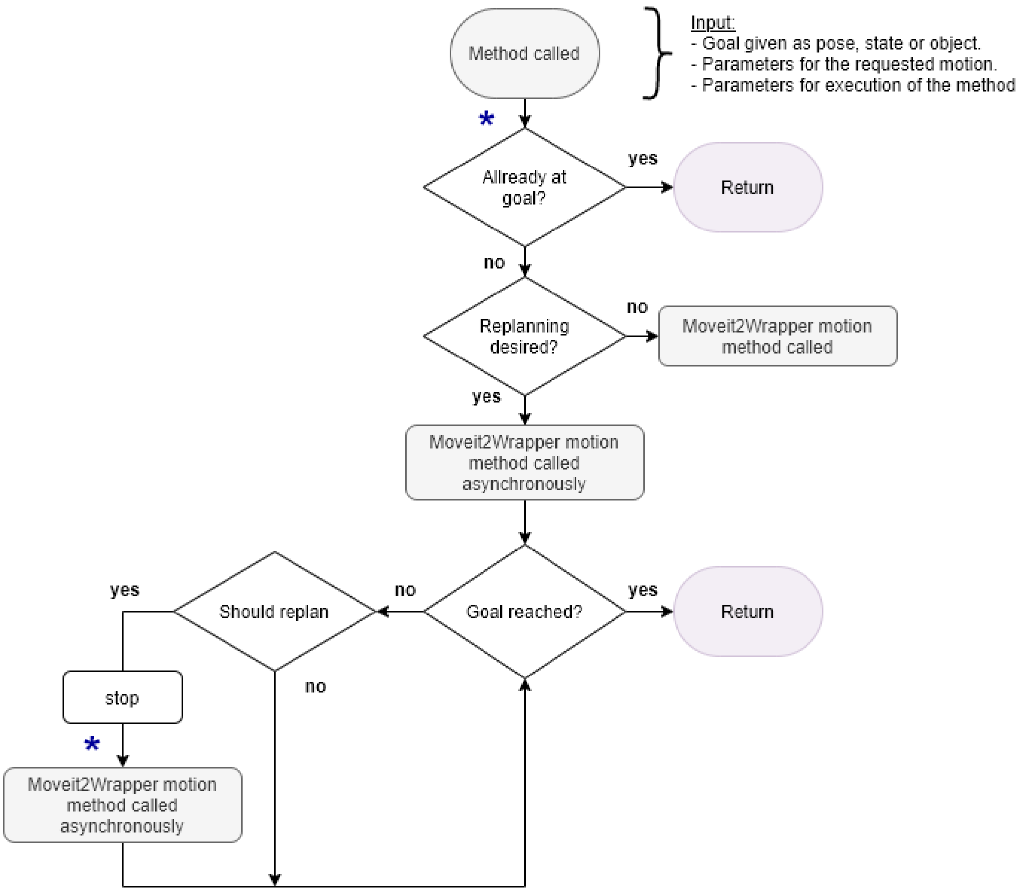

Method called
Input:
- Goal given as pose, state or object.
- Parameters for the requested motion.
- Parameters for execution of the method
*
Allready at goal?
yes
Return
no
Replanning desired?
no
Moveit2Wrapper motion method called
yes
Moveit2Wrapper motion method called asynchronously
Goal reached?
yes
Return
no
Should replan
yes
no
stop
*
Moveit2Wrapper motion method called asynchronously

Figure 6.22: Execution flow of a MotionCoordinator nonlinear pose-to-pose and state-to-state motion.

[1]Can be seen in "event_triggered_replanning_demo.mp4" in the digital appendix.

For linear pose-to-pose motion calling methods, two tests must be passed before `cartesian_pose_to_pose()` is called. The first test checks whether the end-effector of the PlanningComponent is already at the goal. The second test checks whether the goal pose is a valid pose to protect against impossible planning queries. An example of a invalid pose can be a goal pose inherently in collision. After the tests, `cartesian_pose_to_pose()` is called. If no solution was found, and retries are permitted, the motion calling method enters a loop. In this loop, two efforts are made for each retry to change the initial configuration of the PlanningComponent to enable the Cartesian motion method to find a solution. The first effort attempts to find an equivalent configuration which gives the same pose of the end-effector as the initial configuration. This is done using a numerical inverse kinematics solver within the KdlWrapper instance. The solver used is utilizes the Newton-Raphson algorithm presented in Section 2.2.4. In each attempt, to find a equivalent configuration, the inverse kinematic solver iterates through a predefined list of seeds. The seeds can be randomly distributed, or more focused in specific sections of the workspace. If an equivalent configuration is found, the configuration is checked for validity. If deemed valid, the manipulator is moved to the state, and the `cartesian_pose_to_pose()` motion method is called again. If no equivalent configuration was found, the configuration was not deemed valid or no solution was found by `cartesian_pose_to_pose()`, the second effort is initiated.

The second effort, dubbed the fallback effort, is a heuristic approach which leverages the stochasticity of the OMPL-based motion planing pipeline. The planner used, RRT-Connect, plans in configuration space. If the goal is given as a pose, the pose must be transformed to a configuration before planning. MoveIt 2 Beta employs a numerical inverse kinematics solver using the current configuration of the PlanningComponent as seed. Thus, the goal configuration is likely to be found to be different for two different starting configurations. When also considering that two poses close together in Cartesian space may be far apart in the configuration space, it is likely that moving the end-effector to two random poses nearby the original pose before returning to the original pose, may cause the manipulator to be stationed in a different section of the configuration space. The fallback effort employs this technique. If the fallback effort also is unable to find a solution, the `retries_left` counter is decremented and the loop continues until a solution is found or no more retries are allowed. The execution flow is illustrated in Figure 6.23. A blue star indicates where the pose of the object is polled when the goal is given by an registered object.

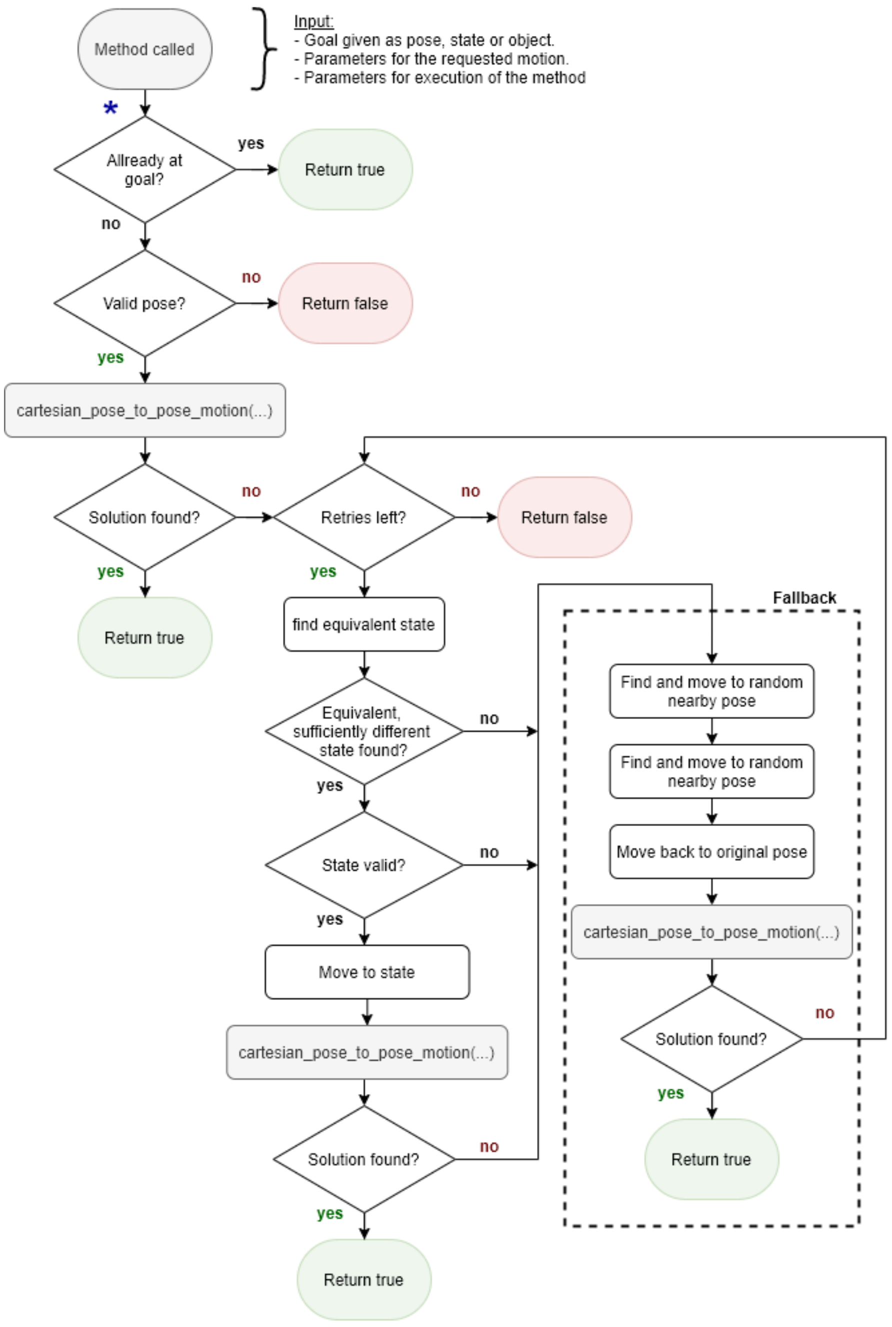

Method called
Input:
- Goal given as pose, state or object.
- Parameters for the requested motion.
- Parameters for execution of the method
*
Allready at goal?
yes
Return true
no
Valid pose?
no
Return false
yes
cartesian_pose_to_pose_motion(...)
Solution found?
no
Retries left?
no
Return false
yes
Return true
yes
find equivalent state
Fallback
Equivalent, sufficiently different state found?
no
yes
Find and move to random nearby pose
Find and move to random nearby pose
State valid?
no
Move back to original pose
yes
cartesian_pose_to_pose_motion(...)
Move to state
Solution found?
no
cartesian_pose_to_pose_motion(...)
yes
Return true
Solution found?
no
yes
Return true

Figure 6.23: Execution flow of a MotionCoordinator motion calling method for linear motion in Cartesian space.

6.3.4 Pose Estimation System

The pose estimation system consists of three ROS 2 modules: a Pose Estimation module which is used for image processing and object detection, the Zivid ROS 2 module which is used to control the Zivid camera, and a Pose Estimation Manager module serving mainly as a user level API while also being responsible for tasks such as coordinate frame transformations and creation of feasible robot grasps from object poses. The pose estimation system, while integrated into the robot system at large, can be seen as completely separate from the robot control architecture. As such, it can be used for a number of object detection tasks serving various purposes. A detailed description of each component comprising the pose estimation system is provided in its own section. A diagram showing a general overview of the system is depicted in Figure 6.24, and a diagram showing the data flow between system components is depicted in Figure 6.25.

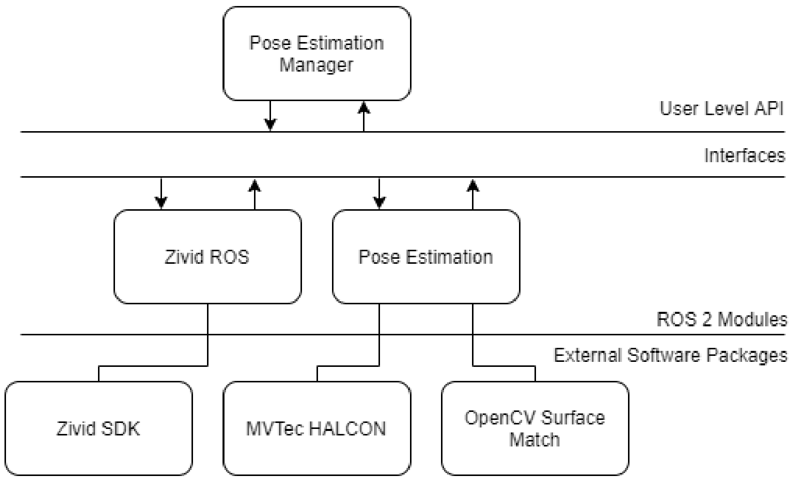


Figure 6.24: Overview of the main components in the pose estimation system, as well as important external software packages used.

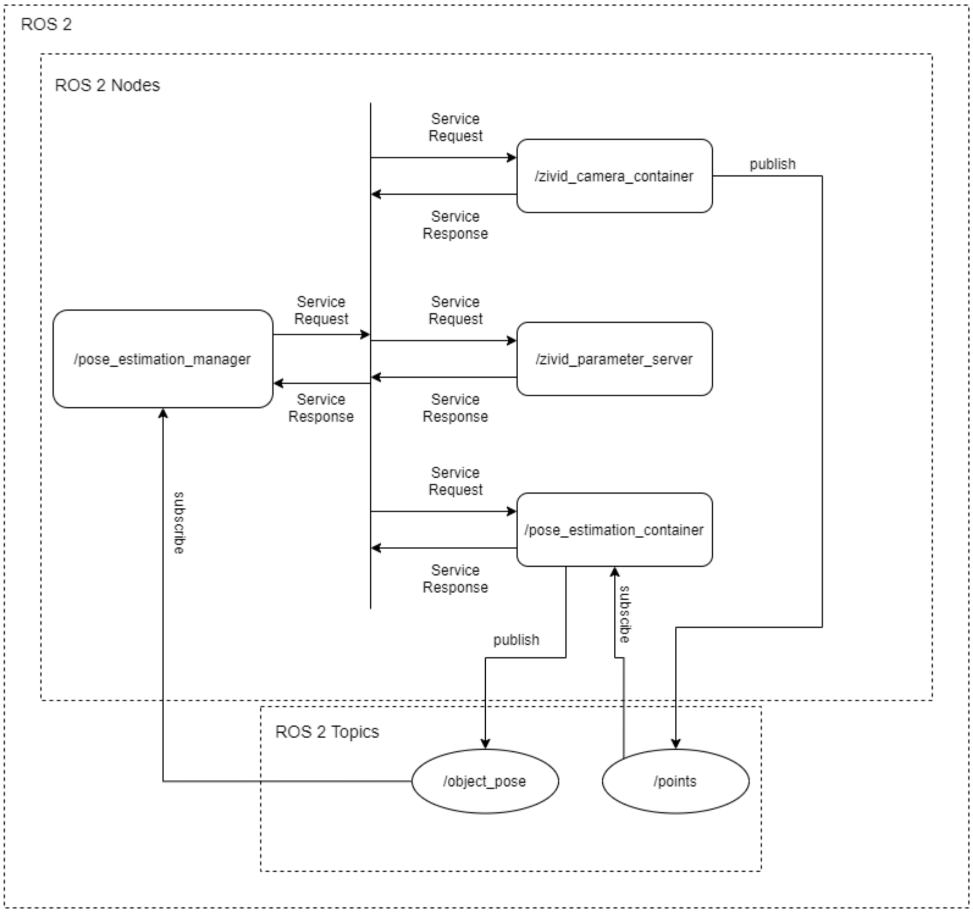


Figure 6.25: Data Flow between the components in the pose estimation system.

Pose Estimation ROS 2 Module

The pose estimation module is implemented using several C++ classes, principal of which is the PoseEstimation class. This class is derived from the ROS 2 LifecycleNode class, thus an instance of the PoseEstimation class is a LifecycleNode. This is done to allow for the use of ROS 2 interfaces, as well as runtime life cycle management. In addition to interfacing functionality, the PoseEstimation class implements methods for preprocessing of point clouds, such as removal of planes and filtering based on a region of interest. Pose estimation functionality is implemented in separate classes. The functionality of which is incorporated in the PoseEstimation class through instantiated members. Currently object pose estimation using point pair feature matching is implemented with the MVTec HALCON computer vision library [44] and OpenCV. The modular design of PoseEstimation was chosen to allow for easy incorporation of additional pose estimation methods implemented in separate classes. Core functionality required to prepare input to each of the pose estimation methods is kept at the top level. The modular design and inter-class relationships can be seen in the UML class diagram in Figure 6.26.

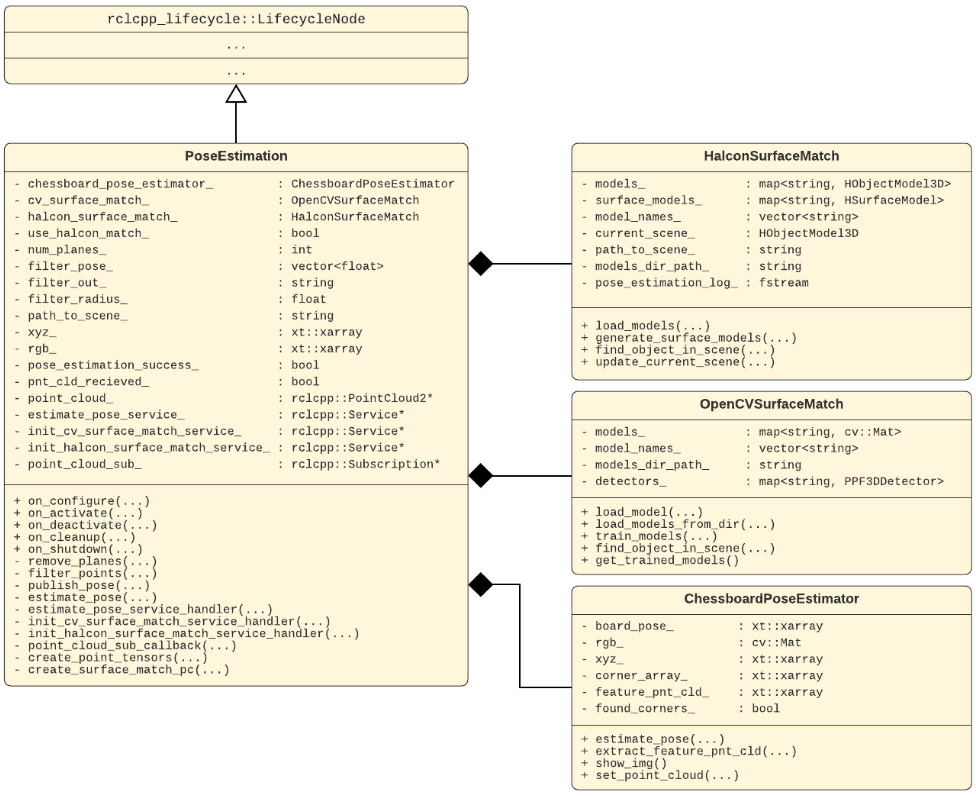
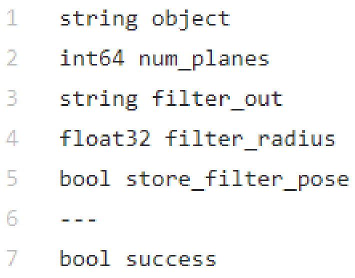

rclcpp_lifecycle::LifecycleNode
...
...
PoseEstimation
- chessboard_pose_estimator_ : ChessboardPoseEstimator
- cv_surface_match_ : OpenCVSurfaceMatch
- halcon_surface_match_ : HalconSurfaceMatch
- use_halcon_match_ : bool
- num_planes_ : int
- filter_pose_ : vector<float>
- filter_out_ : string
- filter_radius_ : float
- path_to_scene_ : string
- xyz_ : xt::xarray
- rgb_ : xt::xarray
- pose_estimation_success_ : bool
- pnt_cld_recieved_ : bool
- point_cloud_ : rclcpp::PointCloud2*
- estimate_pose_service_ : rclcpp::Service*
- init_cv_surface_match_service_ : rclcpp::Service*
- init_halcon_surface_match_service_ : rclcpp::Service*
- point_cloud_sub_ : rclcpp::Subscription*
+ on_configure(...)
+ on_activate(...)
+ on_deactivate(...)
+ on_cleanup(...)
+ on_shutdown(...)
- remove_planes(...)
- filter_points(...)
- publish_pose(...)
- estimate_pose(...)
- estimate_pose_service_handler(...)
- init_cv_surface_match_service_handler(...)
- init_halcon_surface_match_service_handler(...)
- point_cloud_sub_callback(...)
- create_point_tensors(...)
- create_surface_match_pc(...)
HalconSurfaceMatch
- models_ : map<string, HObjectModel3D>
- surface_models_ : map<string, HSurfaceModel>
- model_names_ : vector<string>
- current_scene_ : HObjectModel3D
- path_to_scene_ : string
- models_dir_path_ : string
- pose_estimation_log_ : fstream
+ load_models(...)
+ generate_surface_models(...)
+ find_object_in_scene(...)
+ update_current_scene(...)
OpenCVSurfaceMatch
- models_ : map<string, cv::Mat>
- model_names_ : vector<string>
- models_dir_path_ : string
- detectors_ : map<string, PPF3DDetector>
+ load_model(...)
+ load_models_from_dir(...)
+ train_models(...)
+ find_object_in_scene(...)
+ get_trained_models()
ChessboardPoseEstimator
- board_pose_ : xt::xarray
- rgb_ : cv::Mat
- xyz_ : xt::xarray
- corner_array_ : xt::xarray
- feature_pnt_cld_ : xt::xarray
- found_corners_ : bool
+ estimate_pose(...)
+ extract_feature_pnt_cld(...)
+ show_img()
+ set_point_cloud(...)

Figure 6.26: UML Class diagram for the pose estimation package.

As can be seen in the data flow diagram (Figure 6.25), the PoseEstimation class is contained in the node `/pose_estimation_container`. The ROS 2 interfaces used are services and topics. Services provided by the PoseEstimation node include EstimatePose, InitCvSurfaceMatch and InitHalconSurfaceMatch, the structure of these are detailed below. In addition, the inherited services from the LifecycleNode class are also available.

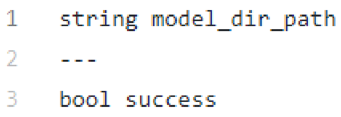
The EstimatePose service exposes the pose estimation functionality of the class to the ROS 2 system. The structure of the EstimatePose service is depicted in Figure 6.27. The request field consists of the name of the object to be found, the number of planes which are to be removed from the point cloud, whether to filter out inliers or outliers of the region of interest, and if a new region of interest should be stored based on the requested object pose. The response field contains a bool which tells the service caller whether the desired object pose was found.



```
string object
int64 num_planes
string filter_out
float32 filter_radius
bool store_filter_pose
---
bool success
```

Figure 6.27: Structure of the EstimatePose service.

The InitCvSurfaceMatch and InitHalconSurfaceMatch services initializes HALCON and OpenCV surface matching respectively. The structure of the surface match initialization services is depicted in fig. 6.28. The request field of the these services contain a string which represents the path to the directory where the 3D models, used for point pair feature matching, are stored. The response field contains a bool which tells the service caller whether the initialization was successful or not.



```
string model_dir_path
---
bool success
```

Figure 6.28: Structure of InitCvSurfaceMatch and InitHalconSurfaceMatch services

The pose estimation node subscribes to the `/points` topic, which contains PointCloud2 ROS 2 messages used to store point clouds. Each time a new point cloud is received, a pointer to the point cloud message is stored in a member variable, ensuring that the most recent point cloud is always available. After an object pose is found by the pipeline, it is published as a PoseStamped ROS 2 message to the `/object_pose` topic, making it available to all nodes subscribing to the topic.

In an effort to reduce the number of points processed by the point pair feature algorithm, two methods of point cloud segmentation are used. One finds planes and removes points close to them, another removes points outside (or inside) a set region of interest. The C++ Point Cloud Library (PCL) [52] is used extensively for this purpose. The plane filtering algorithm is implemented in the `remove_planes()` method. It uses a RANSAC scheme to find planes in the scene, then removes points closer than 5 mm to the plane. The number of planes to be removed from the scene is given as an input to the function call. The function is called recursively until the desired number of planes have been removed. When the pose of an object is to be found, it is possible to specify whether the object's pose should be stored as the region of interest for filtering purposes. This pose is stored in the `filter_pose_` member variable. For any subsequent pose estimation request it is possible to filter out either inliers or ouliers of this region of interest. This functionality is implemented in the `filter_points()` method. This function calculates the distance of each point to the center of the region of interest, if this distance is larger than `filter_radius_` the points are either removed or kept depending on whether outliers or inliers are to be removed. Since both these filtering methods are required to loop over all the points in the point cloud, they result in an increase in processing time of the preprocessing of the point cloud. This is alleviated using parallelization techniques available through OpenMP library [47]. The preprocessing of the point clouds significantly increase the robustness and reduces the processing time of the point pair feature detection algorithm, reducing the total pose estimation time.

HALCON Surface Match

MVTec HALCON is a commercially available image processing software package with a wide array of functionality. The HalconSurfaceMatch class wraps the point pair feature based pose estimation functionality available through the HalconCpp API in a set of convenient methods. An overview of the class can be found in Figure 6.26.

The `load_models()` method takes a string with the path to the location of the directory where object 3D models are stored in a PLY file format, as input. The 3D models in this directory are loaded into a hash table where the key to each object is a string representing the name of the object, extracted from the filename. In the `generate_surface_models()` method, surface models (point pair feature models) are created for each model and stored in another hash table using the same keys. The `find_object_in_scene()` method takes the name of the object to be found as input and uses the `surface_models` hash table to match the correct surface model to the current point cloud scene. The `load_models()` and `generate_surface_models()` are invoked once using the InitHalconSurfaceMatch service, while the `update_current_scene()` and `find_object_in_scene()` is invoked each time a new object is to be found.

The HALCON API allows for the adjustment of matching parameters. The most important of these are "Relative Sampling Distance Model", "Relative Sampling

Distance Scene" and "Key Point Fraction". "Relative Sampling Distance Model" determines the distance between each sampled point on the 3D model of the object, relative to the model size, used for feature generation during training. "Relative Sampling Distance Scene" determines the distance between scene points sampled during matching, relative to the size of the model to be matched. These two parameters directly influence the number of point pair features used for object detection, and has a great influence on performance. Decreasing either relative sampling distances increase the robustness of the matching while being detrimental to the processing time. "Key Point Fraction" determines the number of scene points to be used during matching. Increasing this number influences the matching similar to decreasing relative sampling distances.

The poses found by the point pair feature matching algorithm are based on the local coordinate system of the object, as defined in the 3D model. This information is important when using the poses, for instance when bin-picking. The local object coordinate system has it’s origin at the point where a robot should grip the object. The y-axis is aligned with the axis of symmetry for the object, for objects where such an axis exists.

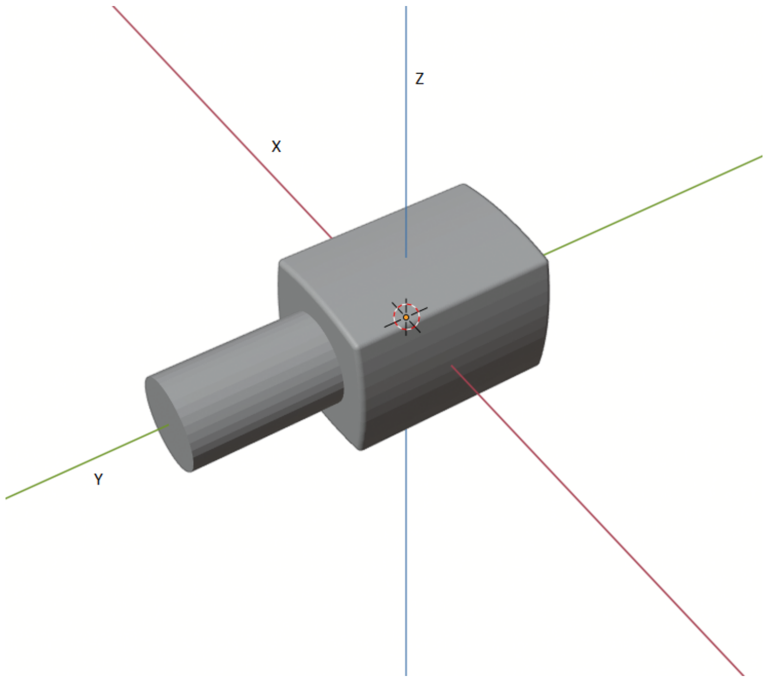


Figure 6.29: Position and alignment of the local object coordinate system for the `nail_polish` 3D model.

Zivid ROS 2 Module

The Zivid ROS 2 module is used to control, and interface with Zivid cameras. It was ported from the ROS module supplied by Zivid AS to ROS 2 by Associate Professor Lars Tingelstad at the Department of Mechanical and Industrial Engineering, NTNU. Implementation details will therefore not be covered, instead an overview of important functionality is presented.

The Zivid ROS 2 module is implemented in the ZividCamera class and functions as an interface between the Zivid SDK hardware interface, and the remainder of the ROS 2 system. The ZividCamera class inherits from the ROS 2 LifecycleNode class. Through ROS 2 services, functionality such as connecting to and disconnecting from cameras, capturing 2D images and 3D point clouds, and adjusting camera settings is provided. Establishing connection to an available camera and disconnecting from it is done automatically when the ROS 2 lifecycle node transitions through different states. The ACTIVATE transition connects to an available camera and the DEACTIVATE transition disconnects from the currently connected camera. Camera settings are updated automatically from the camera parameter server. Camera settings are changed using the SetParameter service provided by the parameter server. The integration of the Zivid ROS 2 module into the ROS 2 pose estimation system is detailed in Figure 6.25.

Pose Estimation Manager

The PoseEstimationManager class encapsulates functionality to control the Zivid Camera module and the Pose Estimation module. In addition, functionality used for coordinate frame transformations. The PoseEstimationManager class is derived from the ROS 2 rclcpp::Node class. Interfaces to the `/pose_estimation_container` and `/zivid_camera_container` are provided by the offered ROS 2 services. Data is transferred using ROS 2 topics.

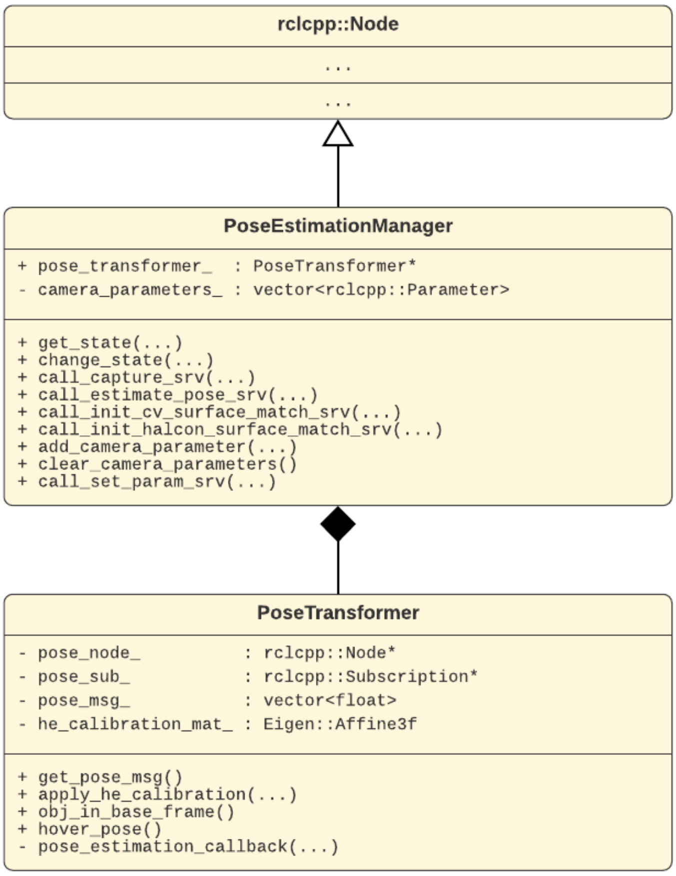
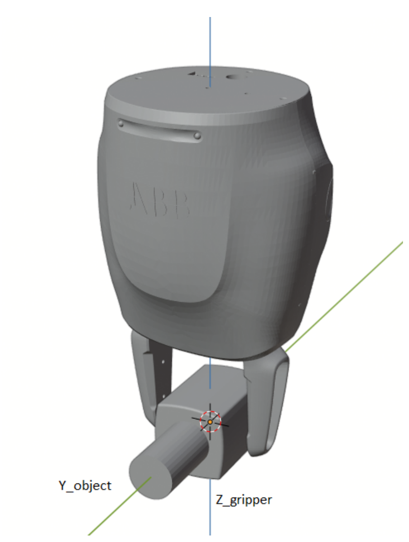

rclcpp::Node
...
...
PoseEstimationManager
+ pose_transformer_ : PoseTransformer*
- camera_parameters_ : vector<rclcpp::Parameter>
+ get_state(...)
+ change_state(...)
+ call_capture_srv(...)
+ call_estimate_pose_srv(...)
+ call_init_cv_surface_match_srv(...)
+ call_init_halcon_surface_match_srv(...)
+ add_camera_parameter(...)
+ clear_camera_parameters()
+ call_set_param_srv(...)
PoseTransformer
- pose_node_ : rclcpp::Node*
- pose_sub_ : rclcpp::Subscription*
- pose_msg_ : vector<float>
- he_calibration_mat_ : Eigen::Affine3f
+ get_pose_msg()
+ apply_he_calibration(...)
+ obj_in_base_frame()
+ hover_pose()
- pose_estimation_callback(...)

Figure 6.30: UML Class diagram for the user level API.

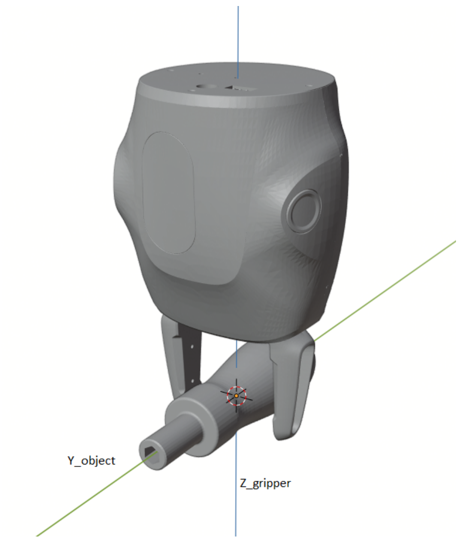
The PoseEstimationManager class provides the user with the `change_state()` method for invoking lifecycle node transitions. These transitions are invoked using the services of the LifecycleNode base of ZividCamera and PoseEstimation. Using the Capture service offered by `/zivid_camera_container`, a point cloud is captured and published as a PointCloud2 message on the `/points` topic. Camera parameters can be changes using the service SetParameters offered by the `/zivid_parameter_server` node. This functionality is provided to the user through the `call_capture_srv()` and `call_set_param_srv()` methods respectively. Parameters to be sent to the parameter server must first be stored locally in the vector `camera_parameters_`, this is done using the `add_camera_parameter()` method. The interface of `/pose_estimation_container` are made available to the user thought the three methods `call_estimate_pose_srv()`,

`call_init_cv_surface_match_srv()` and
`call_init_halcon_surface_match_srv()`.

The PoseEstimationManager obtains the pose of the found object by subscribing to the `/object_pose` topic. The PoseEstimationManager uses an instance of the PoseTransformer class to transform the object pose from the camera coordinate frame to a gripable pose in the robot base coordinate frame. This is done by first transforming the object pose in the camera frame to the robot base frame using the hand eye transformation. This pose can however not be used directly as a end-effector grasp pose. For objects with at least one axis of symmetry, the pose is not uniquely defined. Because of the way the local coordinate systems for each object is defined in the 3D models, the y-axis of the object will always be found correctly by the pose estimator. For all objects, the grip point position is defined by the placement of the local coordinate system. The grasp orientation is found by aligning the y-axis of the the robot end-effector with the y-axis of the object. The z-axis of the end-effector is set to be orthogonal to the y-axis, making sure that the z component of the end-effector z-axis is negative. This ensures that the grip pose is always reachable, as long as it is within the robot workspace. The end-effector alignment is depicted in Figure 6.31.



(a) Gripper alignment on the `nail_polish` model.



(b) Gripper alignment on the `screw_driver` model.

Figure 6.31: Illustration of gripper alignment with the local object coordinate system.

6.3.5 Sensor-less Force Estimation System

The system for continuous estimation of end-effector forces consists of three main components. The first component `etorque_sender` is a task on the robot controller responsible for reading the external joint torques estimated by the robot controller and sending these to the external computer. The next component is the `/e_torque_receiver_node` responsible for receiving the data sent from the robot controller, processing the data stream and publishing the joint torques on ROS 2 topics. Finally, the `/external_force_node` component uses the estimated external joint torques to calculate the corresponding external force in the end-effector frame. An overview of the system components and data flow between them is depicted in Figure 6.32.

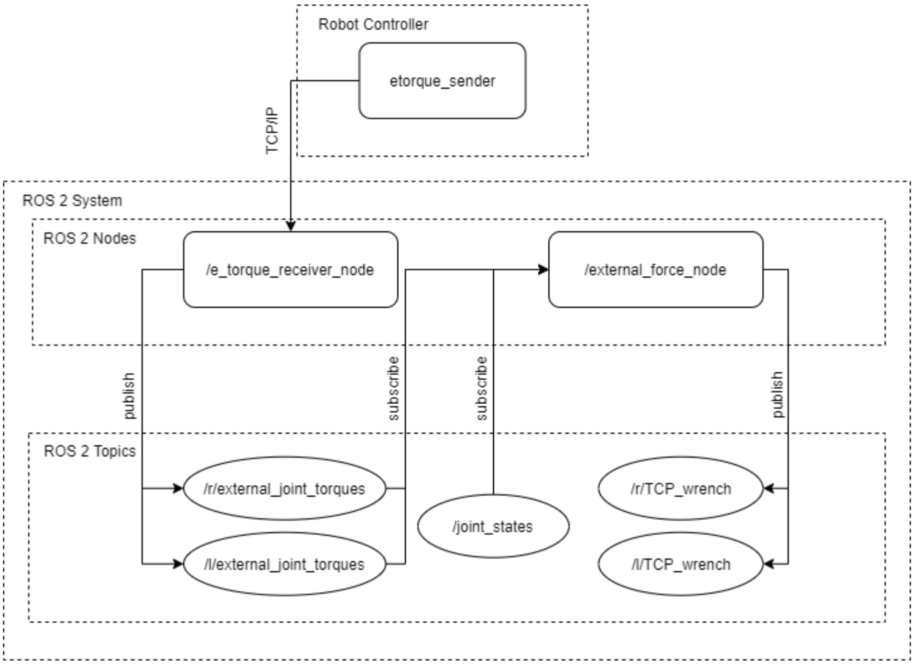


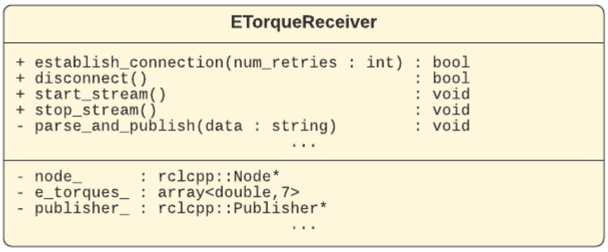
Figure 6.32: Diagram showing the data flow within the external force estimation system.

etorque_sender

`etorque_sender` is the task on the robot controller responsible for reading and transmitting motor torques caused by an external load on an assigned arm. The task hosts a TCP/IP server, reads the motor torques caused by an external load using the TestSignDefine RAPID functionality and transmits the read motor torques with a frequency of about 800 hz. TestSignDefine constructs an internal channel between a variable in the RAPID code and a internal signal defined by a numerical signal ID. The channel is configured with a sampling time used to define the frequency at which the signal is sampled and the variable value is updated. `etorque_sender` uses seven TestSignDefine channels configured to access the "External torque signal" signal of a joint with a sampling time of 0.001 s, meaning seven variables are updated in a multithreaded manner with a frequency of 1000 hz. Once a client has connected to the server, the task enters the transmission loop. In this loop the motor torques are read from the test signals, written into a comma-separated string with start and end delimiters, and sent over the socket. To transmit the motor torques caused by an external load for both arms, two tasks must be running on the robot controller. However, in the version of RobotWare used in this thesis, RobotWare 6.10.01, the number of TestSignDefine channels allowed at once is limited to 12. Motor torques are therefore only transmitted for the right arm.

ETorqueReceiver

ETorqueReceiver is the ROS 2 module communicating with the `etorque_sender` task. The module is implemented using a class containing methods to establish a connection to the robot controller, start and stop receiving motor torques, parsing and publishing received motor torque to a namespaced `/external_joint_torques` ROS 2 topic. The module is launched by a dedicated executable. The class is depicted in Figure 6.33.



```
ETorqueReceiver

+ establish_connection(num_retries : int) : bool
+ disconnect()                            : bool
+ start_stream()                          : void
+ stop_stream()                           : void
- parse_and_publish(data : string)        : void
                        ...

- node_      : rclcpp::Node*
- e_torques_ : array<double,7>
- publisher_ : rclcpp::Publisher*
                        ...
```

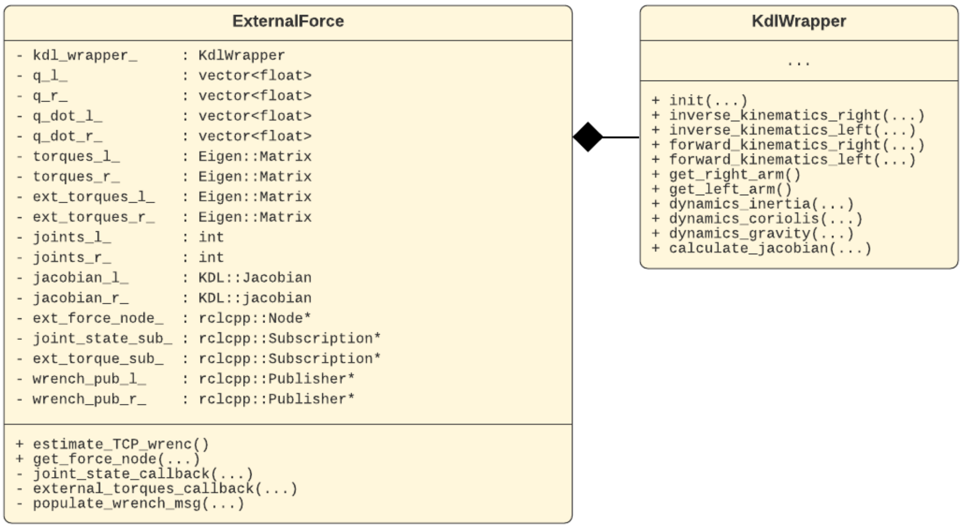
Figure 6.33: Simplified UML diagram of the ETorqueReceiver class.

KDL Wrapper

Orocos Kinematics and Dynamics Library (KDL) [49] is a powerful open source C++ library used to perform direct and inverse kinematics as well as modelling of robot dynamics. Through the use of URDFs, the library can be applied to any robot, since, all kinematic and inertial data is stored within this format. To make use of the KDL software package, a class encapsulating the functionality required in our system has been created. This was partly done in an effort integrate the user API into the overall robotics system, particularly when it comes to the use of standard C++ containers in the place of API specific data types, where appropriate. The exposed API of the KdlWrapper class can be seen in Figure 6.34.

External Force

ExternalForce uses the external joint torques, the current joint configuration, and joint velocities to calculate contact forces or loads applied to the end-effector frame. The dynamic properties of the manipulator in a given configuration is calculated using the KdlWrapper API. Since the external torques estimated by the robot controller includes forces which are due to gravity acting on the manipulator, this must be compensated for before the contact forces can be calculated. This is done by first calculating the torques due to gravity in the given configuration using the `dynamics_gravity()` method of the KdlWrapper. These torques are subtracted from the total external joint torques. The TCP wrench is calculated using Equation (5.11), where the manipulator Jacobian is found using the `calculate_jacobian()` method of the KdlWrapper. The resulting forces and moments are then published on the namespaced topic `/TCP_wrench`, making them available to other nodes in the system.



```
ExternalForce
- kdl_wrapper_      : KdlWrapper
- q_l_              : vector<float>
- q_r_              : vector<float>
- q_dot_l_          : vector<float>
- q_dot_r_          : vector<float>
- torques_l_        : Eigen::Matrix
- torques_r_        : Eigen::Matrix
- ext_torques_l_    : Eigen::Matrix
- ext_torques_r_    : Eigen::Matrix
- joints_l_         : int
- joints_r_         : int
- jacobian_l_       : KDL::Jacobian
- jacobian_r_       : KDL::jacobian
- ext_force_node_   : rclcpp::Node*
- joint_state_sub_  : rclcpp::Subscription*
- ext_torque_sub_   : rclcpp::Subscription*
- wrench_pub_l_     : rclcpp::Publisher*
- wrench_pub_r_     : rclcpp::Publisher*

+ estimate_TCP_wrenc()
+ get_force_node(...)
- joint_state_callback(...)
- external_torques_callback(...)
- populate_wrench_msg(...)
```

```
KdlWrapper
...
+ init(...)
+ inverse_kinematics_right(...)
+ inverse_kinematics_left(...)
+ forward_kinematics_right(...)
+ forward_kinematics_left(...)
+ get_right_arm()
+ get_left_arm()
+ dynamics_inertia(...)
+ dynamics_coriolis(...)
+ dynamics_gravity(...)
+ calculate_jacobian(...)
```

Figure 6.34: UML class diagram for the ExternalForce class and exposed API of the KdlWrapper class.

6.3.6 Bin Picking System

The developed system components are required to function both individually and in an integrated fashion. The implementation of a bin picking system is used as a demonstration and testing platform for the robot system architecture. The bin picking system makes use of the robot control architecture presented in Section 6.3.2, the motion planning architecture presented in Section 6.3.3 and the pose estimation architecture presented in Section 6.3.4.

The bin picking system is implemented using the user level APIs presented in the preceding sections. The execution sequence is as follows. First the robot control and motion planning system is initialized and activated using MotionCoordinator. Then the pose estimation system is initialized by setting Zivid camera parameters, connecting to the camera and initializing the HALCON surface match functionality. At this point, all components of the system architecture are initialized and ready. The first step in the bin picking process is to capture a point cloud of the current scene, locating the pick and place bins using the pose estimation system, and registering them as collision objects in the planning scene. To ensure consistent capture of the scene, the robot arm used for picking moves to a standby state where it does not obstruct the field of view of the camera. For each object to be picked a new point cloud is captured and the pose of the object is found by the pose estimation system. The object is then registered in the planning scene. This object is then picked and placed by the manipulator. The process then repeats with the manipulator moving to it's standby state between each point cloud capture. The program flow is depicted in Figure 6.35, the planning scene during picking is shown in fig. 6.36.

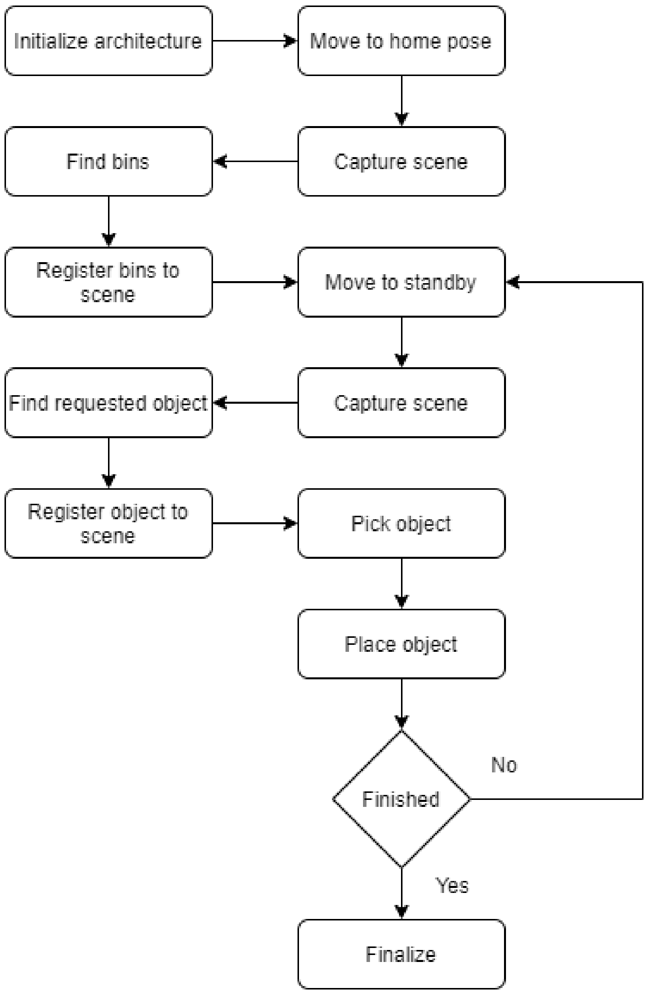


Figure 6.35: Bin picking program flow.

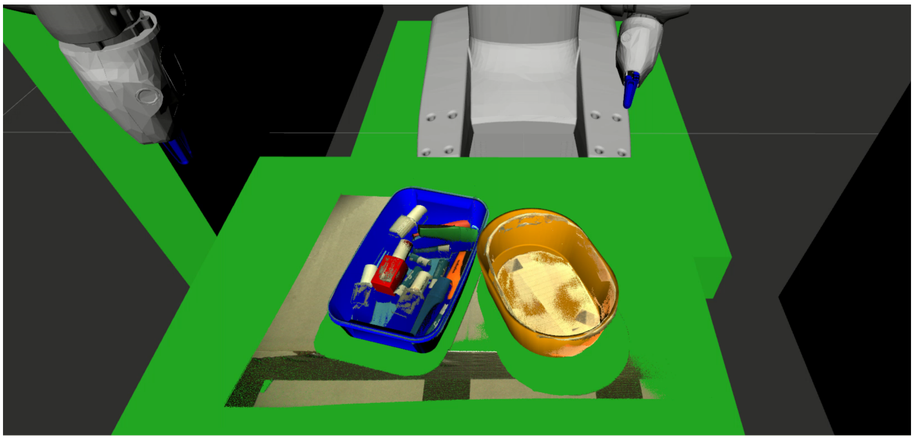


Figure 6.36: Image showing the planning scene during bin picking with the manipulator in it's standby configuration. The modelled static environment surrounding the robot can be seen in green. The pick bin is shown in blue, the place bin in orange and the object to be picked (nail polish) is shown in red. For visualization purposes the point cloud as captured by the Zivid camera is overlaid in the scene. The point cloud is however not part of the planning scene used for obstacle avoidance.

Picking

Picking of objects is implemented in MotionCoordinator in the `pick_object()` method. The method is designed to pick an object registered in the ObjectManager. The method moves the end-effector of the arm to a hover pose above the gripping point on the object to be picked, moves the end-effector in a straight line down to the gripping point on the object, grips the object and returns in a straight line to the hover pose, avoiding collisions with the surroundings. The method is entirely self-contained, meaning as long as the planning scene is updated correctly, no outside logic is needed.

The pick method starts by polling the planning scene for the pose and dimensions of the given object. If the object cannot be found, the pick is aborted and an `PLANNING_SCENE_FAIL` error is returned. If found, the end-effector of the arm is moved to the hover pose over the object, and the gripper is jogged to the width of the object plus an extra predefined margin of error to account for planning scene inaccuracies. Additionally, to account for slight inaccuracies of the pick bin placement, collision checking is disabled between the fingers of the gripper and the bin during the pick. This is done to increase the number of objects eligible for a pick.

After the gripper is opened sufficiently, and collision checking between the fingers of the gripper has been disabled, the gripping pose of the object is checked for validity. If, for instance the gripping pose will cause the gripper base to be in contact with the bin, this test will be failed, the method aborted and a `INVALID_POSE` error returned. If passed, the method attempts to plan a straight line motion down to the gripping pose, given an allowed number of retries to plan a solution. If no straight line solution is found, the pick is aborted and a `LINEAR_PLAN_FAIL` error is returned. If found, the end-effector is moved down to the gripping pose in a straight line and the object is gripped. When the object is gripped, it is attached to the end-effector link in the planning scene. It is thus included in the obstacle avoidance for the manipulator in future motion planning.

In the next step, a straight line motion back up to the hover pose is attempted. Being inside the bin, no retires are allowed. If no solution is found, a nonlinear motion up to the hover pose is planned. When the hover pose is reached, collision checking is re-enabled between the fingers and the bin. Before the `pick_object()` method returns, a check to verify that the gripper contains the object is performed. If the object is dropped, a GRIP_FAIL is returned. The execution flow of a `pick_object()` call is illustrated in Figure 6.37.

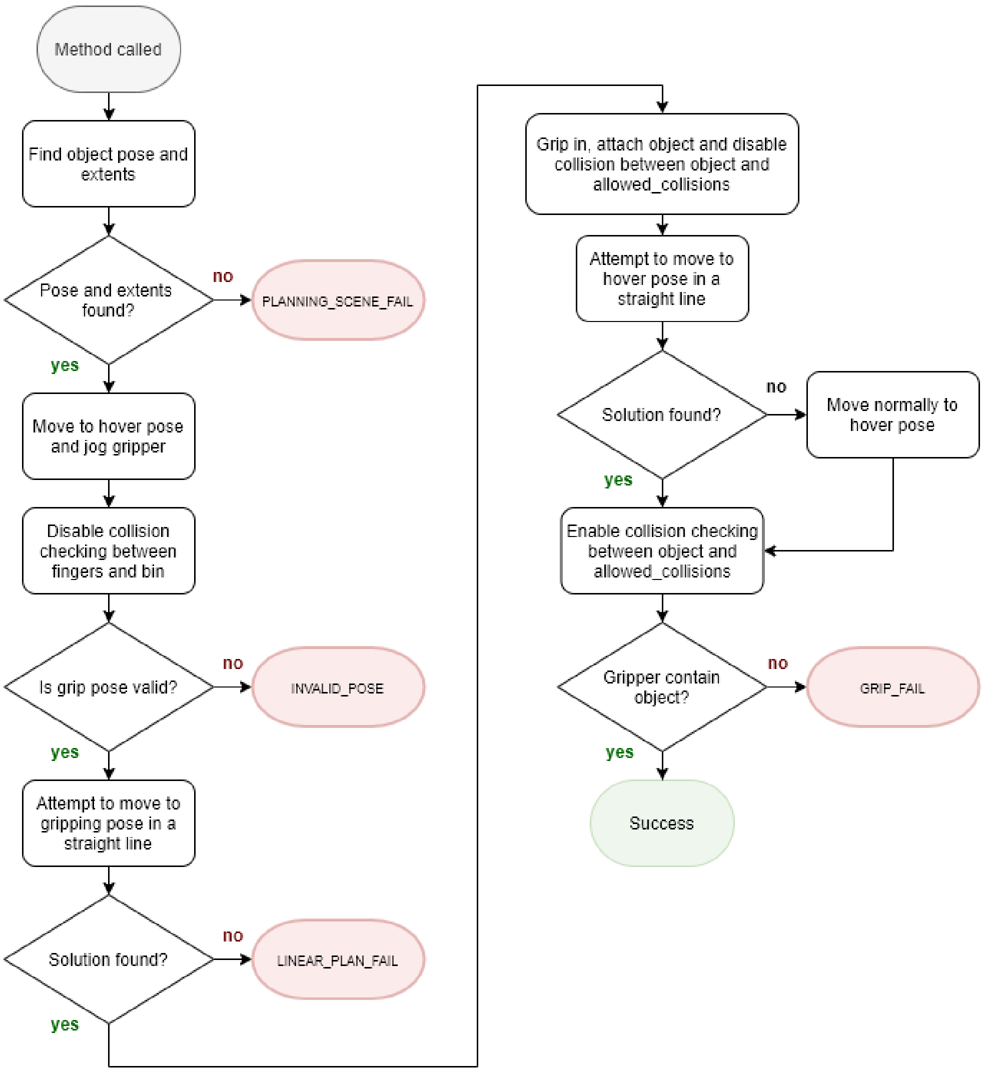

Method called
Find object pose and extents
Pose and extents found?
no
PLANNING_SCENE_FAIL
yes
Move to hover pose and jog gripper
Disable collision checking between fingers and bin
Is grip pose valid?
no
INVALID_POSE
yes
Attempt to move to gripping pose in a straight line
Solution found?
no
LINEAR_PLAN_FAIL
yes
Grip in, attach object and disable collision between object and allowed_collisions
Attempt to move to hover pose in a straight line
Solution found?
no
Move normally to hover pose
yes
Enable collision checking between object and allowed_collisions
Gripper contain object?
no
GRIP_FAIL
yes
Success

Figure 6.37: `pick_object` execution flow.

Placing

Placing of objects is implemented in MotionCoordinator in the `place_in_object()` method. The method attempts to move the end-effector of the arm in a straight line to a hover pose above the place bin. Next, the end-effector is moved down to the drop pose, in the place bin, in a straight line. At the drop pose, the gripper releases the object and attempts to move the end-effector in a straight line back up to the hover pose. Similar to the `pick_object()` method, it is completely self-contained and capable of avoiding collisions.

The method starts by checking whether the gripper contains an object. If dropped, the method returns a `GRIP_FAIL` error and aborts. If passed, the method attempts to plan linear path moving the end-effector in a straight line to a hover pose above the place-object. To avoid unnecessary motions increasing the probability of dropping the object, no retries are given. A straight line motion is preferred, but not required. If no solution is found, a nonlinear path to the hover pose is generated instead.

When the hover pose is reached, the method attempts to plan a straight line motion moving the end-effector to the drop pose. If unable, the object is dropped and detached from the end-effector in the planning scene, before returning success. When the drop pose is reached, the object is dropped and detached. Thereafter an attempts to plan a straight line motion moving the end-effector up to the hover pose is made. If no solution is found, a nonlinear motion to the hover pose is initiated. When the hover pose is reached, the method returns success. The execution flow of a `place_in_object()` call is illustrated in Figure 6.38.

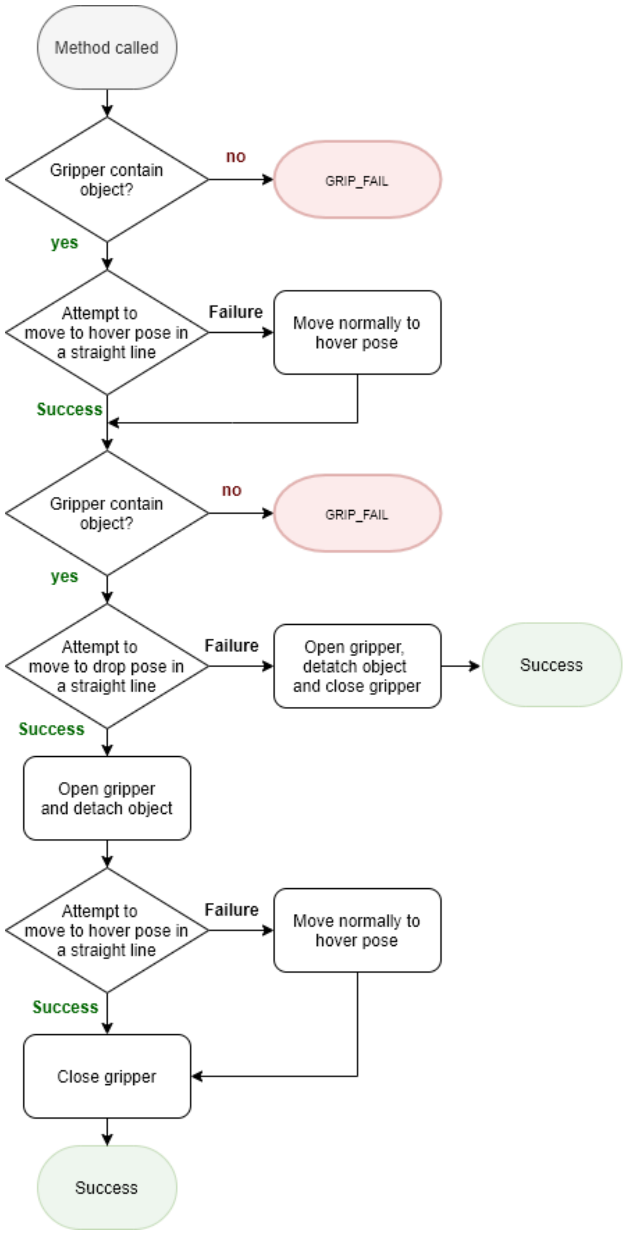

Method called
Gripper contain object?
no
GRIP_FAIL
yes
Attempt to move to hover pose in a straight line
Failure
Move normally to hover pose
Success
Gripper contain object?
no
GRIP_FAIL
yes
Attempt to move to drop pose in a straight line
Failure
Open gripper, detatch object and close gripper
Success
Success
Open gripper and detach object
Attempt to move to hover pose in a straight line
Failure
Move normally to hover pose
Success
Close gripper
Success

Figure 6.38: place_in_object execution flow.

```
enum error
{
    INVALID_POSE = -1,
    LINEAR_PLAN_FAIL = -2,
    GRIP_FAIL = -3,
    PLANNING_SCENE_FAIL = -4
};
```

Listing 6.2: The error enum.

Error Handling

In a robotic system with a high level of autonomy, a number of potential errors are bound to occur during operation. These must be detected and handled in order to avoid undesired system stops. Since the static environment is modelled in the planning scene, and the bins are detected and modelled, the system is able to avoid collisions. Any object held by the manipulator is also collision checked to prevent collisions between the held object and the environment. If the robot drops or fails to grasp an object during picking, this is detected by the system. The system then aborts that pick-place sequence and returns to the standby state for the next attempt. Furthermore, the pose estimation system is able to detect pose estimates likely to be false positives. False positives are most likely to occur when every instance of an object has already been picked. If an object detection is found to be a false positive, the system tries to find a different object. Another failure mode is if the detected object is in a location where the manipulator would collide with it's environment or the pick bin during picking. In this situation, the pick is not attempted and the system continues with the next object. In order for the manipulator to reliably grasp objects, the approach to the object should be following a linear path in the operational space. If such a path cannot be found, the pick is also aborted.

Chapter 7

Experiments

Testing and performance evaluation of the individual system components is an important step in the implementation and deployment of such a system. To this end, a series of experiments were conducted for each component, both in isolation and when integrated as part of a task specific robotics solution for bin picking. Each experiment is presented in a similar manner. The goal of the experiment describes what we want to learn about the system by carrying out the test. The experimental method describes the setup and methodology of the experiment. The evaluation metric describes what quantitative data is to be gathered. Finally, the experimental results are presented.

7.1 Hand-Eye Calibration

7.1.1 Simulations of Hand-Eye Calibration using Point Clouds

Goal

Using generated noisy point cloud data for hand eye calibration is useful to determine the noise-sensitivity of the calibration methods. By generating a synthetic dataset and adding Gaussian noise, the robustness of the methods can be determined by evaluating the calibration error. The simulation results will be used to determine the robustness of 3D point cloud methods for hand eye calibration on noisy datasets.

Method

Noisy point cloud data is generated, simulating an eye-in-hand configuration, by selecting a ground truth calibration object pose in the robot base frame $(T_{bo}^{b})_{GT}$ and a ground truth hand-eye transformation matrix, $(T_{tc})_{GT}$. These were chosen to be

$$(T^b_{bo})_{GT} = \begin{bmatrix} 0 & -1 & 0 & 200 \\ 1 & 0 & 0 & 70 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, (T_{tc})_{GT} = \begin{bmatrix} 1 & 0 & 0 & 50 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 100 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.1}$$

Random robot poses $(T_{bt})_i$ are then generated. The values for the elements of the rotation vectors $k\theta$ are drawn from the normal distribution $N(\mu, \sigma^2) = N(0, \pi^2)$ and the elements for the translation vector of each pose are generated from the normal distribution $N(\mu, \sigma^2) = N(0, (300\sqrt{3})^2)$. It should be noted that the generated poses need not be feasible in the physical system. The corresponding pose of the chessboard in the camera frame $(T_{co})_i$ is calculated as

$$(T_{co})_i = ((T_{bt})_i (T_{tc})_{GT})^{-1} (T_{bo})_{GT} \tag{7.2}$$

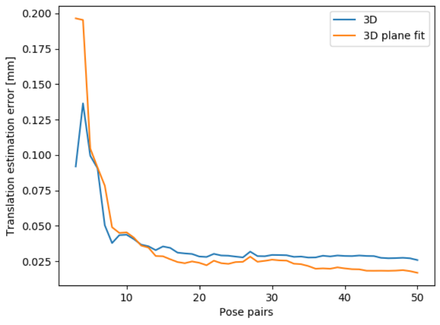
To evaluate the sensitivity of the calibration methods, a point cloud centred in the robot base consisting of $n \times m$ points spaced in correspondence to the center points on the calibration chessboard squares is generated. To each point in this point cloud, a Gaussian noise vector with mean value μ and standard deviation σ is added, resulting in a set of noisy homogeneous chessboard points for each robot configuration, $P_i \in \mathbb{R}^{4\times(n\times m)}$. This point cloud is then transformed to the position of the calibration object using $(T_{bo})_{GT}$ resulting in the object point cloud $(P_o)_i$.

$$(P_o)_i = (T_{bo})_{GT} P_i \tag{7.3}$$

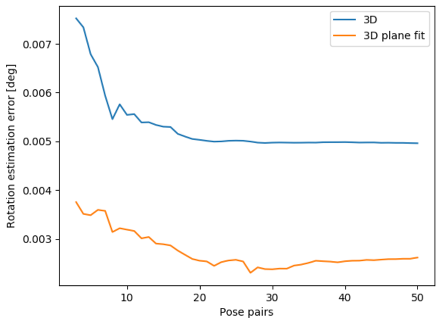
Evaluation Metric

The set of generated noisy calibration points and generated robot poses are used to do hand eye calibration. Error estimates are calculated using Equation (3.24) and Equation (3.25) presented in Section 3.9.1. This is done using an increasing number of pose pairs in order to determine when the error converges. The final translation and rotation errors of each method is used as the performance metric.

Results

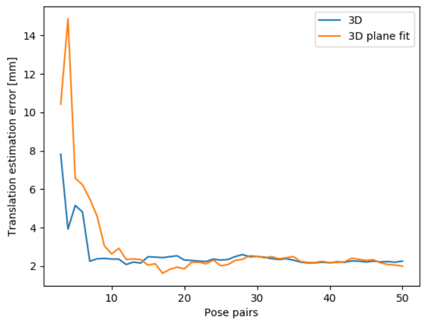


(a) Translation error

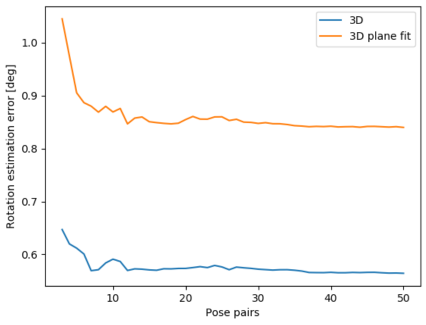


(b) Rotation error

Figure 7.1: Calibration on data with added noise from $N(\mu, \sigma^2) = N(0, 0.01)$



(a) Translation error

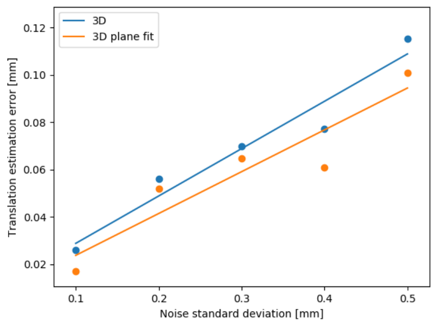


(b) Rotation error

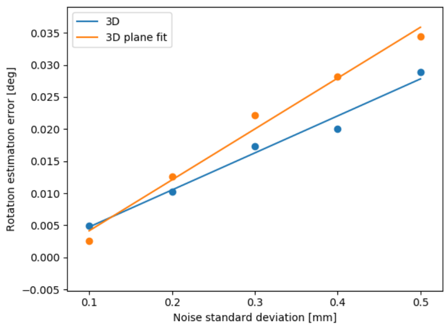
Figure 7.2: Calibration on data with added noise from $N(\mu, \sigma^2) = N(0, 100.0)$

Noise dist. $N(\mu, \sigma^2)$	Trans. error 3D [mm]	Trans. error 3D plane fit [mm]	Rot. error 3D [deg]	Rot. error 3D plane fit [deg]
$N(0, 0.01)$	0.02581	0.01683	0.00496	0.002617
$N(0, 0.04)$	0.05593	0.05195	0.01019	0.01257
$N(0, 0.09)$	0.06975	0.06480	0.01729	0.02219
$N(0, 0.16)$	0.07710	0.06077	0.02007	0.02821
$N(0, 0.25)$	0.11530	0.10083	0.02887	0.03444
$N(0, 1.0)$	0.22951	0.17882	0.05036	0.070979
$N(0, 9.0)$	0.59217	0.68763	0.16361	0.21507
$N(0, 16.0)$	0.85008	0.64449	0.21464	0.29222
$N(0, 25.0)$	1.23648	0.77810	0.27990	0.38845
$N(0, 100.0)$	2.25496	1.99512	0.56434	0.83999

Table 7.1: Final translation errors and rotation errors for 3D and 3D plane fit methods.

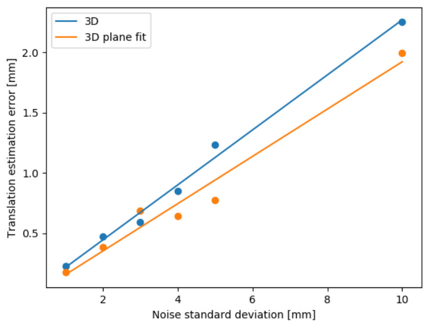


(a) Noise/translation error relationship

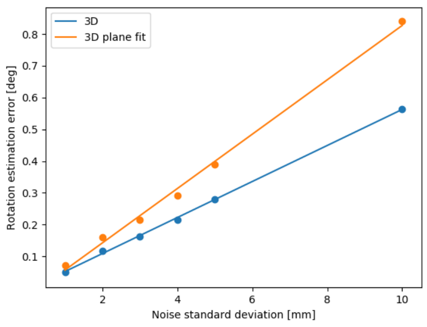


(b) Noise/rotation error relationship

Figure 7.3: Scatter plot and trend line for noise standard deviations $0.1, 0.2, 0.3, 0.4, 0.5$



(a) Noise/translation error relationship



(b) Noise/rotation error relationship

Figure 7.4: Scatter plot and trend line for noise standard deviations $1.0, 2.0, 3.0, 4.0, 5.0, 10.0$

7.1.2 Hand-Eye Calibration of the ABB YuMi and Zivid One 3D camera

Goal

The goal of this experiment is to evaluate the performance of, and compare, hand-eye calibration based on OpenCV extrinsic calibration and based on the two 3D point cloud methods. The outcome of this experiment will hopefully provide some insight into the strengths and weaknesses of each method. In addition to this, determination of the the hand-eye transformation for the robot-camera setup is necessary in order to make further use of the robot vision system.

Method

In our robotic system, the camera is rigidly attached in relation to the robot base frame, as shown in Figure 3.6b. For hand-eye calibration of this setup, the calibration pattern must be rigidly mounted to the robot end-effector frame. To this end, a bracket to hold the calibration pattern was designed and 3D printed. The bracket is depicted in Figure 7.5. The bracket attaches to the robot end-effector using the grippers. To ensure that the bracket does not move when calibrating. The grip point on the bracket is designed to fit exactly to the shape of the grippers.

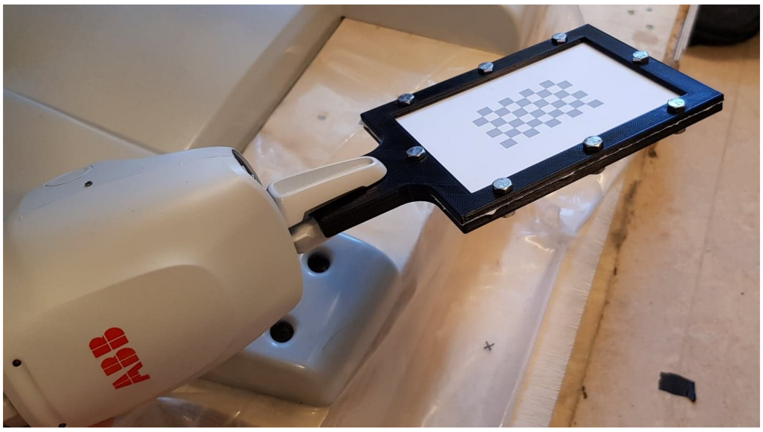


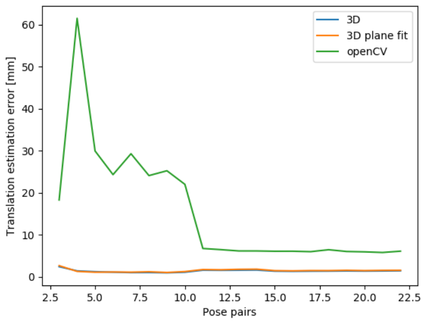
Figure 7.5: 3D printed calibration board rigidly attached to the robot end-effector.

Calibration datasets consist of pairs of robot poses and point cloud images from the Zivid camera. For each pose-image pair in the dataset, the robot was led to a pose and the corresponding image was taken. This was done until a dataset consisting of more than 15 pose-image pairs were gathered. The robot poses were collected using RWS and stored in plain text files. Images were taken using the Zivid C++ API. Two datasets are used to evaluate the calibration methods. For the calibration a python program implementing the methods described in Section 3.5 is used.

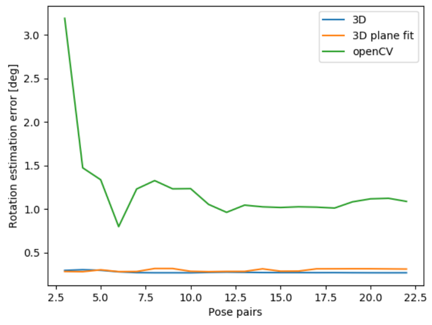
Evaluation Metric

Evaluation of the hand-eye calibration is based on the final translation and rotation error for each method. These are calculated using Equation (3.24) and Equation (3.25). To show how the algorithm converges as the number of pose-image pairs are increased, the error metrics are plotted against the number of pose pairs.

Results



(a) Translation error

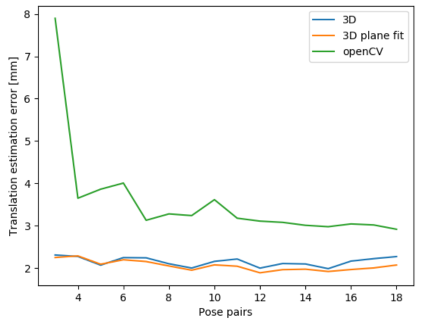


(b) Rotation error

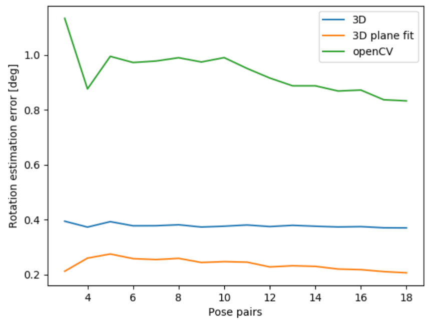
Figure 7.6: Translation and rotation errors for hand eye calibration on dataset 1.

	3D plane fit	3D	OpenCV
Translation error [mm]	1.5414	1.4218	6.0985
Rotation error [deg]	0.3133	0.2708	1.0899

Table 7.2: Final translation and rotation errors for all three method on dataset 1.



(a) Translation error



(b) Rotation error

Figure 7.7: Translation and rotation errors for hand eye calibration on dataset 2.

	3D plane fit	3D	OpenCV
Translation error [mm]	2.0707	2.2702	2.9157
Rotation error [deg]	0.2065	0.3702	0.8327

Table 7.3: Final translation and rotation errors for all three method on dataset 2.

Hand-eye transformation matrix resulting from the 3D plane fit method on dataset 2:

$$T_{bc} = \begin{bmatrix} -0.039 & -0.963 & 0.266 & 189.847 \\ -0.988 & -0.002 & -0.150 & 63.982 \\ 0.145 & -0.269 & -0.952 & 687.990 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.4}$$

7.2 Pose Estimation

Goal

The goal of this experiment is to evaluate the performance of the pose estimation pipeline implemented in the robotics system. The results from this test can be used to make adjustments to improve the overall performance and robustness of the pose estimation.

Method

Data from the pose estimation pipeline were logged continuously, both from stand alone tests and while carrying out bin picking experiments. The pose estimation pipeline was set up to log the total matching time, i.e. the time it takes from the EstimatePose service is called to an object pose is found. Additionally, the match score calculated by the HALCON point pair feature algorithm were also logged. The match score is calculated as the ratio of points sampled from the 3D model, matched to the scene point cloud. This results in the score always being a number between 0 and 1.0, with a score of 1.0 indicating that all the sampled points of an object has been matched with at scene point. Pose estimation time and match score were logged each time the pose estimation service was called. During the pose estimation experiment, ROI filtering was performed removing all outliers. In addition, the ground plane was removed. For pose estimation of the bin objects the whole point cloud was processed.

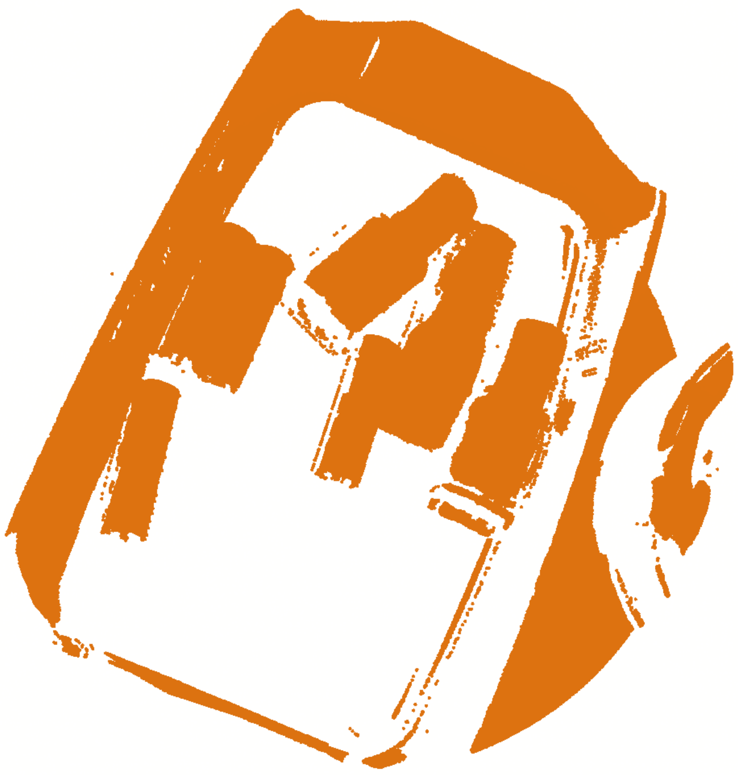


Figure 7.8: Prepossessed point cloud used for matching with region of interest filter and ground plane removed.

Evaluation Metric

The performance of the pose estimation pipeline is evaluated based on matching time for each object, the average score for each object, and the false positive rate for the overall pose estimation system. The accuracy of the point pair feature matching algorithm will not be discussed as this has been extensively investigated previously by other researchers.

Results

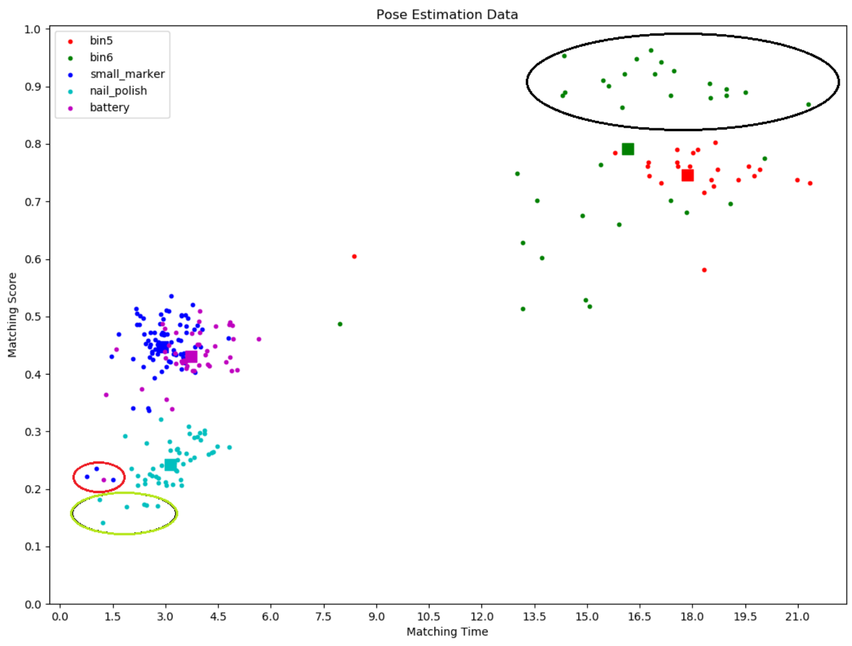
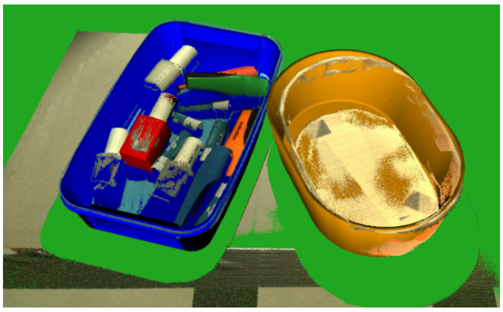


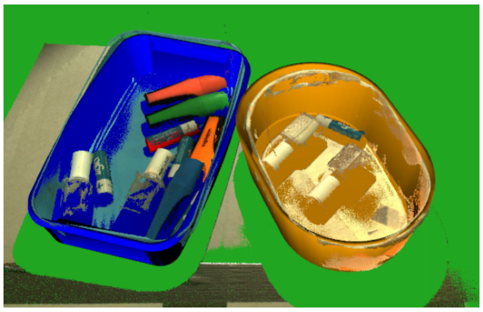
Figure 7.9: Matching time vs matching score for five different objects used in the picking experiments. Mean match scores and pose estimation times are marked by squares. Entries encircled in red were manually identified as false positive matches. Entries encircled in green were identified by the system. The encircled entries of `bin6` were pose estimates carried out on an empty bin, the remainder were taken with objects in the bin.

Object	Mean Time [s]	Mean Score	stddev Time [s]	stddev Score
`small_marker`	2.91	0.45	0.67	0.06
`nail_polish`	3.14	0.24	0.79	0.04
`battery`	3.73	0.43	0.93	0.05
`bin5` (place bin)	17.85	0.75	2.36	0.05
`bin6` (pick bin)	16.15	0.79	2.54	0.15

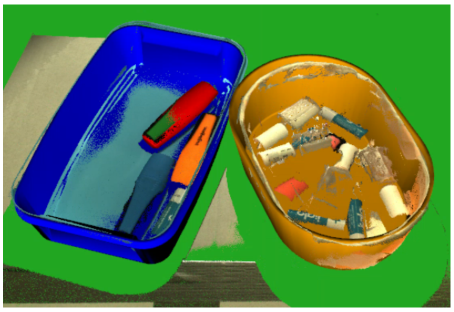
Table 7.4: Per object means and standard deviations for match times and scores



(a)



(b)



(c)

Figure 7.10: Visualization of object matches for `nail_polish` **(a)**, `battery` **(b)** and `small_marker` **(c)**. Matches for `bin5` (orange) and `bin6` (blue) are only valid in **(a)** due to the robot slightly moving them during picking.

7.3 Motion Planning

7.3.1 OMPL-based MoveIt 2 Motion Planning Pipeline

Goal

The goal of the experiment is to evaluate the performance of the OMPL-based MoveIt 2 motion planning pipeline. The outcome of the experiment can hopefully provide insight into how the pipeline can be used to reliably produce solutions in a larger system.

Method

Two experiments were conduced. In the experiments, the pipeline was asked to move the end-effector of the right arm from its home configuration to a randomly generated pose. The randomly generated poses are generated by adding a random translation and rotation to a base pose in front of YuMi. The magnitude of the translation was 15 cm. The rotation was about either the z or y axis with an angle randomly selected between 20 and 50 degrees. Planning was conducted in a static planning scene consisting of obstacles to be avoided. The random poses were generated in an unobstructed section of the workspace. The base pose and the planning scene can bee seen in Figure 7.11. 100 repetitions were conducted in both experiments. In the first experiment, no additional planning attempts were allowed. In the second, three retries were allowed upon a planning failure, meaning a total of four planning attempts were permitted. In both experiments, the pipeline will timeout after 2.0 seconds.

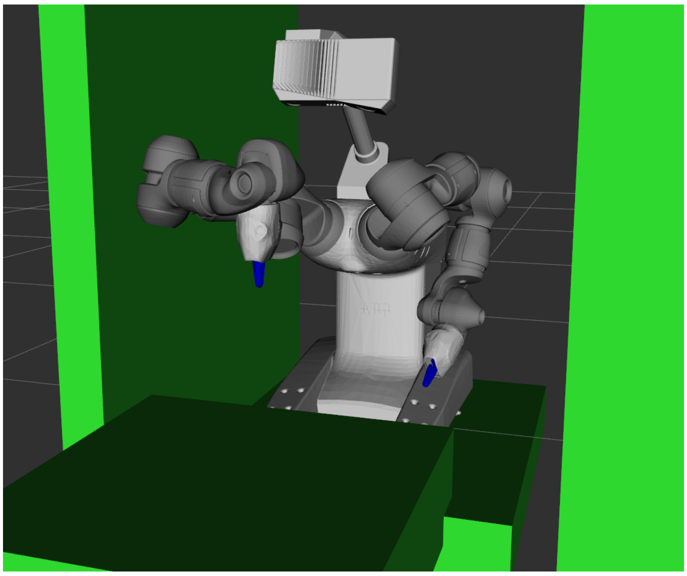


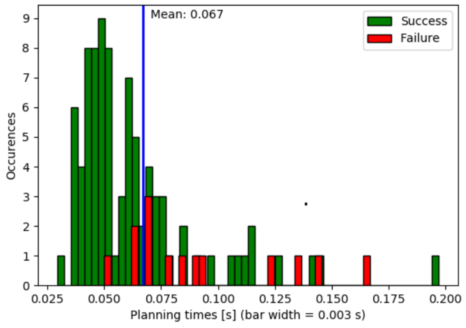
Figure 7.11: The base pose and the planning scene as seen in Rviz2.

A response time is defined as the time between planning was initiated, and a bool indicating planning success or failure was returned. Retries are included in the response times. The experiment was conducted using the simulated backend.

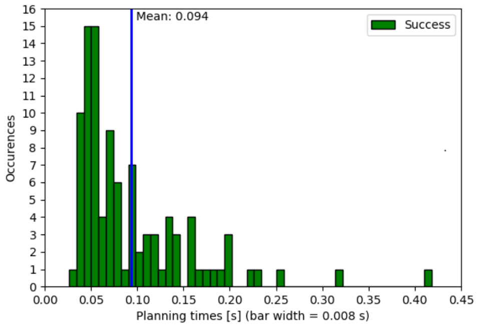
Evaluation Metric

The performance of the motion pipeline is evaluated based on it's success rate, mean response time, worst case response time (WCRT).

Results



(a) Experiment A, 0 allowed retries.



(b) Experiment B, 3 allowed retries.

Figure 7.12: Recorded response times of the OMPL-based motion planning pipeline.

Experiment	Allowed Attempts	Mean Time [s]	WCRT [s]	stddev Time [s]	Success Rate
A	1	0.0670	0.1943	0.0305	86 %
B	4	0.0938	0.4172	0.0644	100 %

Table 7.5: Statistics and success rates measured in the two experiments.

7.3.2 Linear Cartesian MoveIt 2 Motion Planning Pipeline

Goal

The goal of the experiment is to evaluate the performance of the linear Cartesian MoveIt 2 motion planning pipeline. The outcome of the experiment can hopefully provide insight into how the pipeline can be used to reliably produce solutions in a larger system.

Method

Three experiments were conducted. In the first experiment, the pipeline was tasked with moving the end-effector of the right arm linearly in Cartesian space from a randomly generated starting pose, to a pose with the same orientation 10 cm away in the negative z-direction of the base frame and back to the starting pose. Only one planning attempt for each motion was allowed. In the second experiment, the pipeline was tasked with moving the end-effector of the right arm linearly in Cartesian space from a randomly generated starting pose, to a pose with the same orientation 10 cm away in the positive y-direction of the base frame and back to the starting pose. Only one planning attempt for each motion was allowed. In the third experiment, planning to the same pose as in the second experiment was conducted. Three retries were allowed. Due to the motions performed during the retries, the planning times for this experiment were not recorded. In all three experiments, the random starting pose was generated by in the manner described in Section 7.3.1. If planning failed toward the goal, a return solution was not attempted. 50 repetitions were conducted in all experiments.

A response time is defined as the time between planning was initiated, and a bool indicating planning success or failure was returned. The experiment was conducted using the simulated backend. The parameters used with the pipeline is presented in Table 7.6.

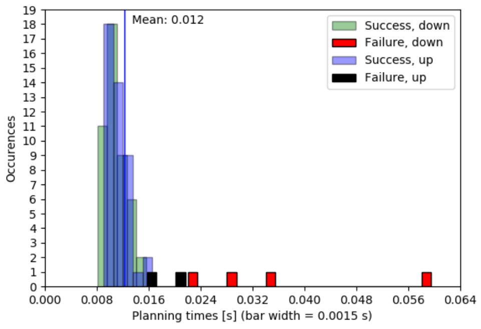
Parameter	Value
Cartesian max step	0.002
Joint threshold factor limit	6

Table 7.6: Parameters used with the linear Cartesian motion planning pipeline.

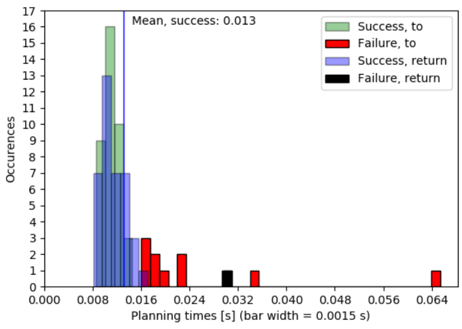
Evaluation Metric

The performance of the linear Cartesian MoveIt 2 motion planning pipeline is evaluated based on its success rate planning to the goal, returning from the goal and consecutively to the goal and back, it's mean response time, and it's worst case response time (WCRT).

Results



(a) Experiment 1, linear motion along the Cartesian z-axis.



(b) Experiment 2, linear motion along the Cartesian y-axis.

Figure 7.13: Recorded response times of the linear Cartesian motion planning pipeline.

Experiment	Num samples	Mean Time [s]	WCRT [s]	stddev Time [s]
1	96	0.0122	0.0585	0.0060
2	89	0.0131	0.0645	0.0072

Table 7.7: Mean time, standard deviation and worst case response time of the experiments.

Experiments	Success Rate			
	To	Return	To and Return	Overall
1	92 %	95.5 %	88 %	93.7 %
2	78 %	97.4 %	76 %	86.5 %
3	96 %	N/A	N/A	96 %

Table 7.8: Planning success rates of the experiments.

7.4 Bin Picking

Goal

The goal of the bin picking experiment is to evaluate the performance of the implemented robotics system in a realistic component handling task. Of particular interest is the evaluation of the error handling mechanism in the system, and where improvements can be made in order to increase the system robustness.

Method

The experimental setup constitutes the robotics system as described, two bins, and three different components which are to be picked from one bin and placed in another. The three components are `battery`, `small_marker` and `nail_polish`. The pick-bin and place-bin were both placed randomly in the scene and must be detected by the pose estimation system and registered in the planning scene, before objects are picked. A total of 9 runs, consisting of 10 required picks, were conducted using the system parameters in Tables 7.9 and 7.10. These parameters were tuned between runs in an attempt to reduce the pick time and cycle time.

Parameter \ Experiment	1,2,3	4	5	6,7,8,9
Num Planes Filtered	0	1	1	1
ROI Filtering	ON	ON	ON	ON
Max joint speed [deg/s]	57.3/57.3	57.3/57.3	57.3/57.3	180/400
Max joint acc. [deg/s^2]	57.3/57.3	57.3/57.3	57.3/57.3	180/180
Linear Speed Scaling, pick	1	1	0.3	1
Standby Configuration	Home	Home	Home	Alternative

Table 7.9: System parameters for each of the experiments. Max joint velocities and accelerations are given for the first 4 joints and and the last 3, [first/last]. The alternative configuration is slightly outside the camera's field of view, closer to the bins than the home configuration.

HALCON Setting \ Experiment	1,2,4,5,6,7,8,9	3
Rel. Sample Dist. Model	0.03	0.02
Rel. Sample Dist. Match	0.03	0.02
Key Point Fraction	0.5	0.5

Table 7.10: HALCON settings used in the bin picking experiment.

Evaluation Metric

Evaluation of the bin-picking capabilities of the robotics system can be divided into two categories, robustness and performance. Robustness is measured by the ability of the system to handle errors when they occur in an effort to avoid undesired stops. Performance is measured by average pick time i.e. the time the system uses on a successful pick-place operation, and by average cycle time i.e. the average time taken to perform a pick-place operation when error handling is necessary. Under ideal operating conditions, when few errors occur, average cycle time should be close to average pick time.

Results

Planned Picks	90
Successful Picks	89
Attempts	106

Table 7.11: Number of planned picks, successful picks and the number of attempted picks during the experiment. The number of attempts represent the sum of successful picks, aborted picks due to error handling and outright failed attempts where the system does not detect the error when it occurs.

Error Type	Number of Failures	Number of Failures Handled
Grip Failure	3	3
Invalid Pose	11	11
Linear Planning Failure	0	0
Pose Estimation Failure	3	2
Total	17	16

Table 7.12: Failure modes for the 17 unsuccessful attempts. One false positive object match was not detected.

Experiment	Grip Failure	Invalid Pose	PE Failure
1	-	4	-
4	-	1	-
5	-	3	1
6	1	-	-
7	1	-	-
8	1	-	1
9	2	-	1

Table 7.13: Distribution of error types for experiments with aborted picks.

Experiment	1	2	3	4	5	6	7	8	9
Avg. Detection Time	14.50	15.05	20.65	5.79	5.14	4.35	4.35	4.33	4.72

Table 7.14: Average processing time for the pose estimation pipeline, including the point cloud acquisition time, for each experiment.

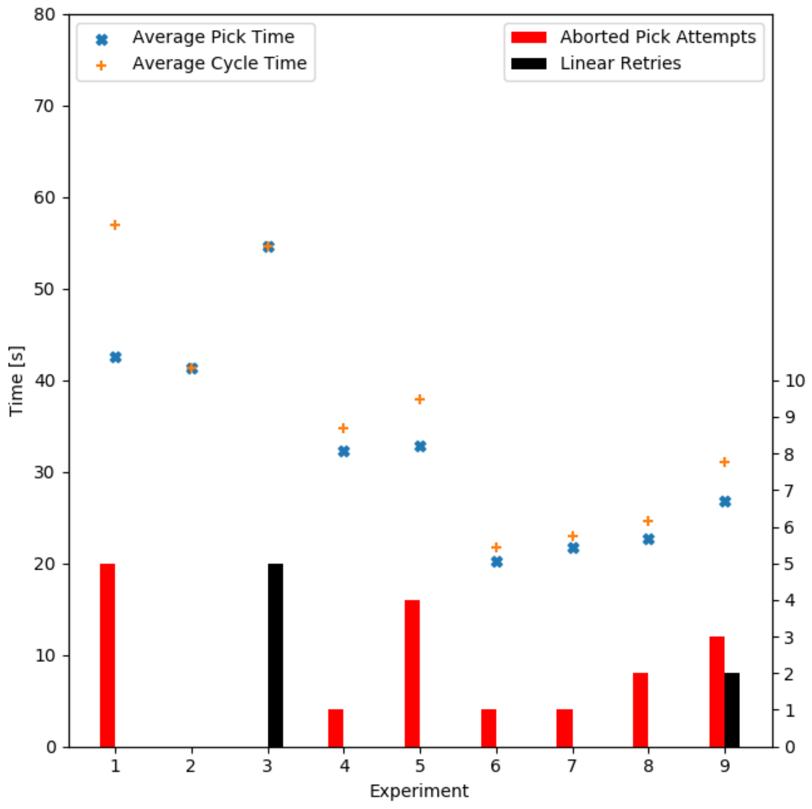


Figure 7.14: Average pick and cycle times. The number of aborted picks are shown as red bars, the number of linear planning retries accumulated over each the experiment are shown as black bars. These are numbered on the right hand side vertical axis.

Videos of experiment 7 and 9 can be found in the digital appendix.

7.5 External Force Estimation

Goal

The goal of this experiment is to test and evaluate the system for estimation of external forces acting in the end-effector frame. The results are to be used to determine whether or not the force estimation system can be used for applications such as force control in it's current state.

Method

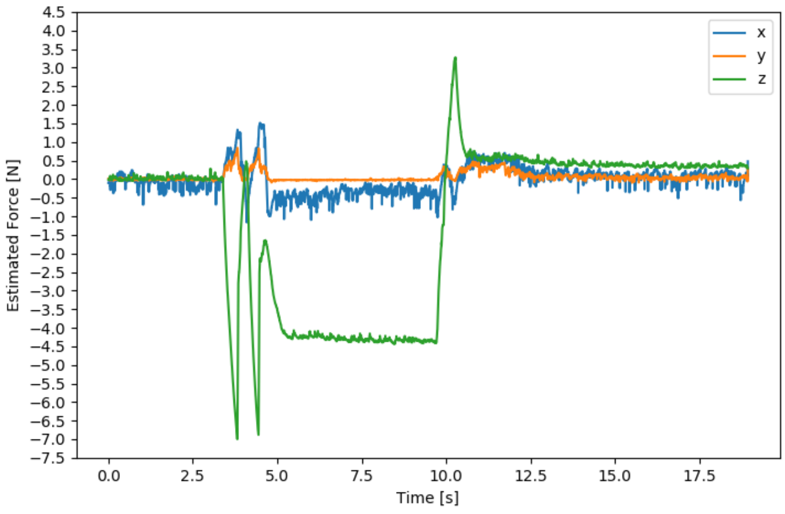
While the robot arm is stationary various forces is applied to the robot end-effector. These forces are either applied by hand or by placing an object with a known mass in line with one of the axes of the end-effector frame. The forces applied by hand are mainly to observe the response of the system under load, as the direction and particularly the magnitude of the force vector is uncertain. The test using a known mass should yield results which can be used to evaluate the systems ability to accurately estimate forces on the system.

Evaluation Metric

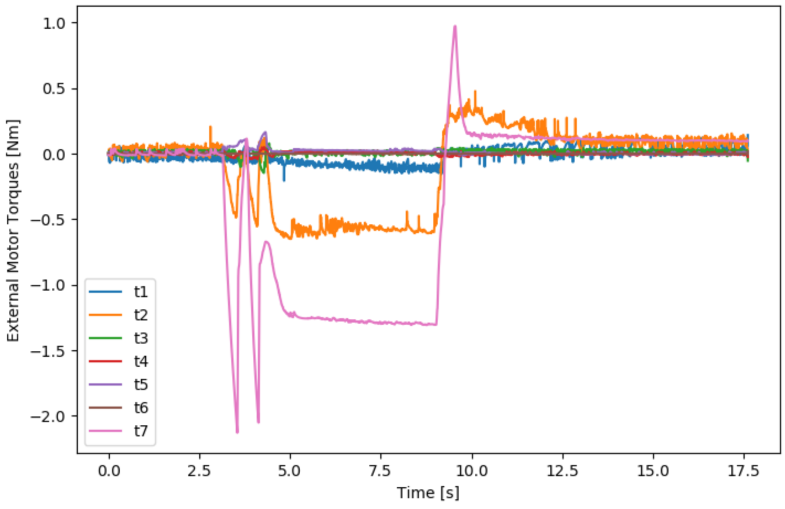
Evaluation of the force estimation system is based on the accuracy of the force estimate when loaded with a known mass, and the it's ability to detect forces in different directions and decompose the force vector in each axis of the end-effector frame. The system is also evaluated by it's practical usability for force control applications.

Results

Unfortunately we were unable to acquire the correct mass and inertial data for the manipulator used in the experiments. The consequence of this is that the system is not able to correctly compensate for the contribution to external torques due to gravity. This results in incorrect offsets for all the measured torques which again results in incorrect offsets for the force estimates. The system was therefore only evaluated based on the difference in measured force for each axis. For visualisation purposes each data entry is centered at 0.0 to show the deviation from the initial force/torque values.



(a)



(b)

Figure 7.15: External force estimate **(a)** and external torque measurements **(b)** when placing an object weighing 0.498 kg roughly in line with the end-effector z-axis.

Estimated force vector from Figure 7.15a, with the end-effector under load.

$$\mathbf{F} = (-0.5, 0.0, -4.4)\text{N} \tag{7.5}$$

$$||\mathbf{F}|| = 4.42\text{N} \tag{7.6}$$

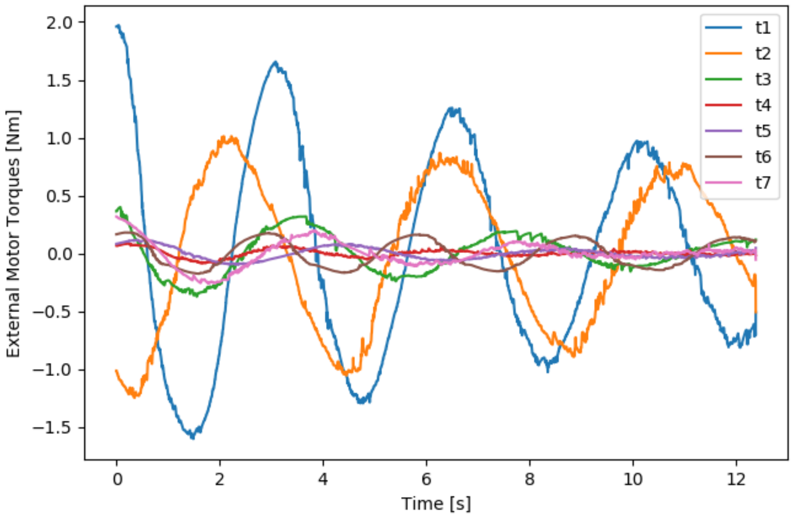
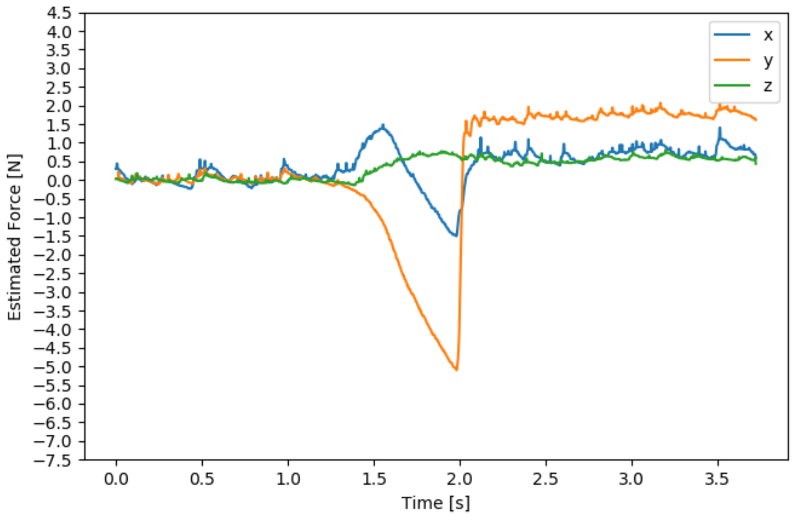
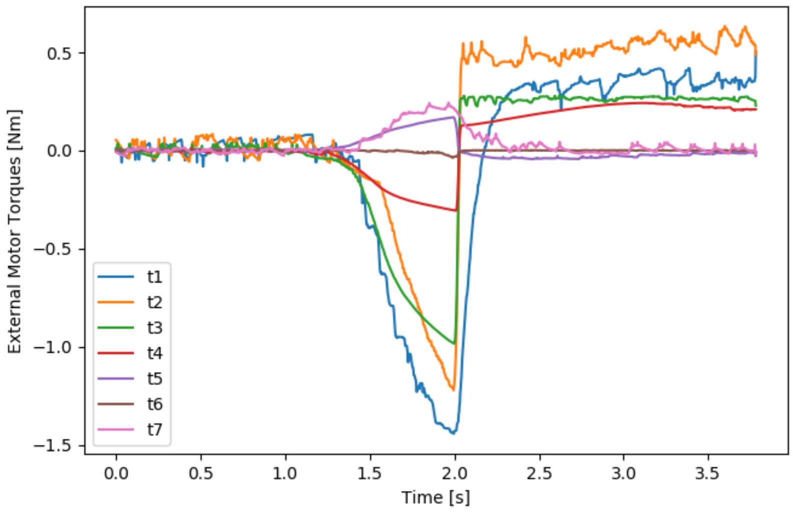


Figure 7.16: Oscillations present in the system after motion of the manipulator.



(a)



(b)

Figure 7.17: External force estimate **(a)** and external torque measurements **(b)** when applying a force in the xy-plane of the end-effector frame.

Chapter 8

Discussion

8.1 Hand Eye Calibration Using Point Clouds

Simulation of hand-eye calibration using generated noisy point clouds show that both the 3D point cloud correspondence method and the 3D plane fit method produce relative camera poses accurate enough for the hand-eye calibration algorithm to converge. Both methods also result in similar convergence rates. There does however appear to be a difference in which error metric is minimized by the two methods. Observing the plots of the final errors for both methods in Figures 7.3 and 7.4 it is clear that the 3D plane fit method results in a lower translation error at all noise levels. The error trend lines for the two methods also diverge slightly as the noise level increases, indicating that the position estimate of the 3D plane fit method is also more robust when it comes to increases in noise levels compared with the 3D correspondence method. The opposite is observed in the plots of rotation error vs. noise level. Here the 3D correspondence method is consistently more accurate than the 3D planar method. The trend lines also diverge, indicating that the orientation estimate of the 3D plane fit method is more sensitive to increases in noise level than the 3D correspondence method.

It is difficult to pin-point the exact reason for the observed results. Both methods calculate the relative pose of the camera, for use in hand-eye calibration by considering the translation of each calibration point cloud's centroid. The average translation of the centroid between the ground truth point cloud and the centroid for each calibration point cloud is used as the metric for calculating the error in the calibration for both methods. It would be reasonable to expect that the method which can achieve the highest orientation accuracy would also achieve the highest position accuracy, since the optimal rotation matrix is used to optimize the translation vector in the hand-eye calibration algorithm used. This is however not consistent with the simulation results.

The results of hand-eye calibration on datasets gathered using the YuMi robot and Zivid One camera, show that all three methods tested allow for convergence of the hand-eye calibration algorithm on the available datasets. The point cloud methods do however converge faster than the OpenCV based method and result in more accurate calibrations. Our calibration method had, on average, 2.46 times lower translation error and 3.3 times lower rotation error compared to the OpenCV 2D method. The performance of the two 3D methods were fairly comparable, with the 3D plane fit method performing better on dataset 1 and the 3D point cloud correspondence method performing better on dataset 2. The observed difference in translation and rotation accuracy of the two methods in the simulations were not corroborated in the calibration on real data. On the real datasets the method with the lower rotation error also had the lower translation error, this is more in line with what one might expect based on the implemented hand-eye calibration method. It is also unlikely that the noise characteristics are similar for the simulations and real data. The simulations do not consider inaccuracies in the absolute positioning of the robot end-effector or slight movements in the calibration board relative to the end-effector during dataset collection, which are certainly present in calibration of the real robot system. The practical application of the resulting hand-eye calibration matrix was tested in the bin picking experiment, confirming that the calibration error is sufficiently small for the manipulation of small components using 3D vision.

8.2 Pose Estimation System

The pose estimation pipeline performed well, with the vast majority of attempted matches resulting in a correctly identified object and pose. This was observed using the visualization of the planning scene in rviz2, as exemplified in Figure 7.10. A few false positive matches were not identified as such by the pipeline. This is because we somewhat naively, through initial testing using the `nail_polish` object, determined that a lower score limit for a true positive match should be 0.2. This limit was however too low for the remainder of the objects. The method used by HALCON to calculate match scores results in different expected scores for each object type. The expected score depends on the percentage of the object's surface likely to be visible to the 3D camera during pose estimation. For the the `nail_polish` object, the percentage of it's surface captured in the point cloud is likely to be between 20% and 30%. For the `battery` and `small_marker` objects, this number is likely to be somewhere between 40% and 60%. This explains why a lower score limit of 0.2 was selected based on initial testing with the `nail_polish` object.

To lower the false positive rate, a per object minimum match score should be implemented. This score can be determined using the mean score and the score standard deviation of true positive matches. Since the false positive matches were observed to have a significantly lower score than the true positive matches, the lower limit can be selected fairly conservatively to ensure that all true positives are identified as such. It is however the case that undetected false positives can be very detrimental in robot applications as they may result in unexpected behaviour. For this reason it is prudent to err on the side of caution when selecting the lower score limit. If new objects are introduced to the system, an initial lower score limit should be based on the percentage of the surface likely to be visible by the camera at any time. In our experiment, it should be noted, false positives only occurred when the object searched for by the pipeline was not present in the scene.

The match time observed in our experiment is consistent with what is expected from the performance of point pair feature matching algorithms. The comparatively longer times for the two bins were likely due to the fairly large size of these objects, resulting in a larger amount of point pair features being sampled. They were also matched against an unfiltered point cloud, as no region of interest is determined at the time of matching, and removal of the ground plane resulted the determination of the bin's height over the base being poorly conditioned. In this case we consider this detriment to performance as an acceptable cost for more robust matching. The large variation in score for `bin6` was due to the variation in visible surface when it was loaded with components compared to when it was empty. The bin was found consistently even when it was loaded with components, since a relatively large percentage of the surface was still visible to the camera.

8.3 Motion Planning

The experiments testing the performance of the OMPL-based MoveIt 2 motion planning pipeline indicate a clear improvement in success rate when given multiple planning attempts. In the first experiment, the pipeline was able to solve 86% of the planning requests without any additional attempts. Over the 100 requests, a mean response time of 67 ms was measured, with a worst case response time of 194 ms and a standard deviation of 30 ms. In the second experiment the pipeline was given 3 additional planning attempts, which resulted in all 100 planning requests being successfully solved. Over the 100 planning request, a mean response of 94 ms was measured, with a worst case response time of 417 ms and a standard deviation of 64 ms.

The experiments indicate that additional planning attempts will elevate the success rate for a given planning request. This can partly be explained by the stochastic nature of the employed RRT-Connect motion planner, and partly due to how a pose goal is transformed to a configuration in the pipeline. RRT-Connect uses a randomized sampler, which cause the two trees to often grow completely differently even for planning attempts towards the same goal configuration. In addition, the pose goal may be realized as a different joint configuration by the pipeline, as the kinematic redundancy of YuMi enable a multitude of different configuration to yield the same end-effector pose. Due to the variability of the determined goal configuration and the grown trees, re-calling the planner upon failed planning seem to be an efficient strategy to increase the reliability of the pipeline.

It is also noted the mean response time increases with 3 allowed replanning attempts. As additional attempts are included in the response times, this is expected. Even so, for both experiments, most response times seem to hover around 50 ms. As the first experiment showed, it can be expected that 86% of the requests will be solved in the first attempt in this region of the workspace. With this in mind, it is expected that most successful planning attempts will be within roughly the same range of response times. The increased mean, worst case response time and standard deviation is explained by the more computationally expensive 'new' successes produced by allowing retries. The random poses generated were both in close vicinity of each other, and in an unobstructed section of the robots workspace. It is therefore unknown whether these results can be extrapolated to more difficult planning scenarios. This notwithstanding, the results suggest the pipeline is reliable enough to be used in a larger system.

The experiments testing the performance of the linear Cartesian MoveIt 2 motion planning pipeline suggest the pipeline may perform differently depending on the direction of the desired the linear Cartesian motion. In the first experiment, motion in the negative z-direction of the base frame was desired. 50 attempts at planning a motion 10 cm in this direction and back to the start pose were conducted. Over the 96 response times, a mean response time of 12 ms, with a worst case response time of 58 ms and a standard deviation of 6 ms was measured. In the second experiment, motion in the positive y-direction of the base frame was desired. Likewise the first experiment, 50 attempts at planning a motion 10 cm in this direction and then back to the start pose were conducted. This time, owing to a lower success rate in planning the first motion to the goal, 89 response times were measured. The mean response time was found to be 13 ms, the worst case response time was measured to 64 ms and the standard deviation 7 ms. The first experiment recorded success rates of 92% to the goal, 95.5% returning from the goal, a 88% success rate at finding a solution to the goal and back and an overall success rate of 93.7% over all attempts. The second experiment recorded lower success rates in most categories, a 78% success rate to the goal, a 97.4% success rate returning from the goal, a 76% success rate at finding a solution to the goal and back and an overall success rate of 86.4%.

There appears to be little difference in the response times of a successful planning attempt between the two experiments. The increase in mean response time, worst case response time and standard deviation appear to be caused by the increased number of unsuccessful planning attempts. Even though both experiments plan linear motion in unobstructed sections of the robots workspace, planning in the negative z-direction of the base frame show a 14% better success rate to the goal. This may be caused by the mechanical layout of the robot. The arm of the robot is required to cross in front of its torso, which possibly is a section of the workspace in which it is more difficult to plan linear Cartesian motions. Returning from the goal, the motion planning pipeline displays a similar success rate in the experiments.

The linear Cartesian motion planning experiments indicate that when a solution for a short linear Cartesian motion from an initial pose to a goal pose is found, a returning linear Cartesian motion from the goal pose back to the initial pose can also be expected to be found. This is somewhat unsurprising as the Cartesian interpolator uses a numerical inverse kinematic solver using the last configuration as seed, meaning the exact same path as the path to the goal is likely to be found in reverse. It is noted that this result was achieved for a short motion in an unobstructed part of the robot's workspace. It is not certain that this will be the case in general. Furthermore, while the returning motion is reliably found, the motion to the goal does not deliver the same guarantees. In the last experiment, the planning to the goal was conducted allowing for 3 retry iterations. This increased the success rate to 96%, indicating that if given additional retries, the motion planning pipeline should be able to reliably find solutions. It is noted that the retries are very time consuming due to the additional motions performed.

The two sets of experiments testing the two MoveIt 2 motion planning pipelines highlights the different time-complexities and reliabilities of the pipelines. The OMPL-based pipeline appears to plan on average 6-7 times slower than the linear Cartesian pipeline. The large difference in time-complexity can be explained by a combination of two factors, the different approaches employed to plan paths, and the difference in operational space distance between the start and goal pose. Path planning in the linear Cartesian pipeline is less complex than in the OMPL-based pipeline as the path is predefined as the straight line between two points. The Cartesian interpolator transforms the pose waypoints on the straight line in operational space to the configuration space using inverse kinematics. This process is less complex than the path planning of RRT-Connect in the OMPL-based pipeline. In the experiments, the operational space distance is also shorter for the linear Cartesian pipeline. This results in fewer states being collision checked, which is a very computational expensive operation.

8.4 External Force Estimation

While initial tests of the external force estimation module appeared to yield promising results, further testing indicated that the capability of the system was lacking. The main problem with the overall system is the presence of sinusoidal oscillations in the external torques estimated by the robot controller. We are unable to determine the reason behind these observations as we do not have access to the software implementation responsible for the torque estimates. The oscillations are dampened out over time, as can be seen in Figure 7.16. After sufficient time has passed the system is at rest and further investigation of the force estimation can be conducted.

In the test where a known force of 4.88 Newtons was placed in line with end-effector z-axis, the direction of the estimated force is as expected. The estimated force had a component in the z-direction with a magnitude of 4.4 Newtons and a component in the x-direction with a magnitude of 0.5 Newtons. The force vector has a magnitude of 4.42 Newtons, which is 9.5% less than the applied force. An interesting observation is that the estimated force component in the z-direction does not return to it's initial baseline after the load is removed, instead it reports a force of 0.5 Newtons in the opposite direction of the applied load. The difference in estimated force in the z-direction during and after the load is applied is 4.9 Newtons. This value is within a surprisingly small margin of error of the applied force. The phenomenon of external joint torques not returning to the same value after loading is also observed in Figure 7.17a. One likely reason for the difference in resting torque before and after loading might be differing static friction levels, which are not accounted for in our model.

It is clear that while the dynamic modelling implemented in the ExternalForce class, calculating forces from joint torques appears to work as intended, the external torques estimated by the robot controller do not lend themselves well to this type of application. The oscillating behavior after motion and unpredictable behavior after application of forces, render the current system unsuitable to any application requiring a reasonable level of accuracy. The external torque values are used by the robot controller to detect collisions, it is however unclear how, or if, these values are processed for this purpose. Due to the limited payload of YuMi, the implementation of a system for measuring forces acting on the end-effector frame without the use of external sensors is worth pursuing further. To this end it would be interesting to implement a system for estimation of external torques based on the derivations in Chapter 5.

8.5 System Integration and Bin Picking

While there is room for improvement, the developed robot system was able to reliably pick and place components from a cluttered bin. Over nine experiments, a total of 17 errors were encountered. Of these, 16 were correctly categorized and successfully handled in accordance to the four defined failure modes of the system. One error was not detected, and resulted in a system failure. The failure was caused by a false positive match by the object pose estimation system. Different parameters were employed in the motion planning and pose estimation systems over the nine experiments. The first three experiments were performed using default MoveIt 2 joint speed and joint acceleration limits, and no planes were removed from the point cloud in the pose estimation pipeline. In these experiments, the detection time was roughly 12 seconds, the average successful pick time was 46.2 seconds, while the average cycle time was 50.9 seconds. Five picks were aborted in experiment 1, in experiment 2 and 3 none were aborted. In experiment 3, a total of five linear retries were conducted to find a linear Cartesian motion down into the pick bin. The first two experiments utilised the same settings for the pose estimation pipeline, and unsurprisingly the average detection times recorded were similar. In the third experiment, the relative sampling distances was decreased to 0.02 in HALCON. This increased the detection time with about 5 seconds, with no clear improvement in accuracy.

For the remainder of the experiments, the ground plane in the point cloud was removed in the pose estimation pipeline. This resulted in three times faster object pose estimation. In experiment 4, faster pose estimation in combination with only one aborted pick, and no retry iterations being necessary during linear Cartesian motion planning, resulted in a significantly lower average pick and cycle time, respectively 32.2 and 34.8 seconds. In experiment five a reduced downwards linear Cartesian motion speed scaling was tested while using the same configurations as in experiment four. This resulted in a slighter higher pick and cycle time. In experiment five, four picks were aborted, this was caused by invalid poses where the pick would have resulted in a collision. In experiments 6, 7, 8 and 9, the joint speed and joint accelerations limits were increased, and the standby configuration was changed to the alternative configuration. The average pick and cycle time of these experiments was 22.8 and 25.4 seconds respectively. In experiment 6 and 7 one pick was aborted, and no retry iterations were necessary. Thus, these exhibit similar pick and cycle times. In experiment 8 and 9, 2 and 3 aborted picks occurred, leading to higher cycle times. In experiment 9, two retry iterations were performed, further increasing both the cycle and pick times.

From the data captured in the bin picking experiments, it appears that the retry system employed for planning of linear Cartesian motions is very time consuming. This was highlighted in experiment 3. While no picks were aborted in experiment

3, it appears the conducted retry iterations when finding a linear Cartesian motion were almost equally time consuming as the aborted picks suffered in experiment 1. It is noted that the detection time in experiment 3 was 5 seconds longer than in experiment 1 and 2, and thus accounts for 5 out of the 13 second increase in cycle time from experiment 2 to experiment 3. The effect of retry iterations can also be seen in experiment nine. While useful to reduce the number of aborted picks, the retry iterations do not decrease the cycle time significantly. As was seen in experiment 3, it's most distinct effect was keeping the pick and cycle times similar. Aborted picks have the opposite effect, as the average cycle time will increase without affecting the average pick time. In industrial settings, the cycle time is of greater importance than the pick time. Knowing the average cycle time is necessary to determine the number of components that can be picked or handled during a unit of time, or how long it takes to pick or handle a set number of objects. Additionally, the average cycle time should be stable to provide operational predictability. The additional time added by retry iterations, can be reduced by changing the implementation of the linear Cartesian retry system. The motions performed to find a different manipulator configuration can instead be simulated, resulting in a single motion to the new configuration.

The processing time of the pose estimation pipeline proved to be rather sensitive to changes in the size of the input point cloud. Removing the ground plane in addition to performing region of interest filtering greatly reduces the total number of points. With fewer points, the processing time was reduced to about 32% of the processing time when the ground plane was not removed. The HALCON settings used in our experiments were based on recommended values [25]. Our values were selected with the matching robustness in mind. Parameter values which result in fewer points being processed would likely have resulted in a slight improvement in processing time, at a detriment to robustness. The parameters used in our tests resulted in correct matches when the object was present in the scene, within a reasonable time frame for the employed matching algorithm. The performance of the point pair feature matching algorithm, while not being state-of-the-art for pose estimation, was able to process imaging data at a reasonable rate. The average for the pick objects in experiments 4-9, excluding the image acquisition time, was 3.36 seconds. For deployment of this system in an industrial environment, for instance as part of an assembly line, this relatively high processing time is not ideal. The main advantage of the pose estimation system in it's current configuration is the ease with which new components can be introduced and recognised by the system. Fast reconfiguration for new components allows for shorter turnover times for the system.

Several improvements can be envisioned for the bin picking system. The frequent occurrence of invalid end-effector poses in the picking phase was due to the length of the fingers being very similar to the depth of the pick-bin. This made it difficult for the gripper to grasp objects which were close to the walls of the bin, as this would have resulted in a collision between the gripper base and the top edge of the bin wall. This problem can be resolved by using a set of fingers longer than the depth of the bin. With this change it is likely that the majority of the aborted pick attempts due to invalid poses could have been carried out. Attempts which were aborted due to the gripper dropping the object could have been avoided if the grippers had a higher coefficient of friction than the ones currently used. The single false positive match not detected by the pose estimation pipeline must be eliminated to ensure safe and reliable operation. To this end, the score limit for detection of false positives should be adjusted in accordance with the discussion in Section 8.2. The OMPL-based motion planning pipeline using the RRT-Connect motion planner does not optimize for direct paths. The efficiency of the bin-picking system can be improved by altering the OMPL-based motion planning pipeline to produce more direct paths. This can be done by changing to a more direct planning algorithm.

The interaction between the system components worked as intended. The component's ability to detect errors and report these to the system integration layer ensured that the behaviour of the system was predictable, reducing the probability of undesired system stops. The architecture is applicable to several industrial manufacturing applications involving the handling of small components. In it's current configuration, the system is able to avoid known obstacles, and has the ability to detect new obstacles within cameras relatively narrow field of view. To improve the system's ability to work safely in close proximity with humans, additional sensors could be added to detect obstacles within the whole robot workspace. Because of the modular base-architecture this can be done without major reworking of the current code base. The current system has the ability to perform event triggered re-planning of motions when changes in the planning scene occurs. Thus, if a module tasked with using sensor data to model and monitor obstacles in the planning scene is developed, the current system can be employed to avoid obstacles in a dynamic scene.

Chapter 9

Conclusion and Further Work

9.1 Conclusion

This thesis has presented methodology and implementation details for a developed system for external control of an ABB YuMi dual arm industrial robot. The developed system integrated a state-of-the-art Zivid One 3D camera for object pose estimation together with a motion planning system capable of collision avoidance, allowing for autonomous manipulation of components in a partially structured environment. The robot was externally controlled using the Externally Guided Motion (EGM) and Robot Web Services (RWS) interfaces. The motion planning system was implemented using the initial MoveIt 2 Beta release. The pose estimation pipeline was implemented using point pair features for matching 3D models to point clouds. The system components were integrated using ROS 2. In addition, a method for extrinsic camera calibration using point clouds for use in robotic hand-eye calibration has been developed. A system for estimation of external end-effector forces was also implemented.

The implemented system components were tested both individually and in an integrated system configuration. The pose estimation pipeline was able to reliably find the pose of objects from point clouds. The pipeline was tested on three objects with differing geometry. The average processing time was 3.36 seconds. The motion planning system consists of linear Cartesian and OMPL-based motion planning pipelines. The linear Cartesian motion planning pipeline was found to have a success rate of 96% with an average planning time of 12.65 ms. The OMPL-based motion planning pipeline was found to have a success rate of 100% with an average planning time of 84.4 ms. The method developed for hand-eye calibration was tested against an OpenCV based 2D method. Our calibration method had, on average, 2.46 times lower translation error and 3.3 times lower rotation error compared with the OpenCV method. The resulting hand-eye transformation was used successfully in a robot vision system. The performance of the implemented external force estimation system was found not to be sufficient to merit it's use in any application requiring accurate force measurements. This was due to noisy and inaccurate measurements of external joint torques, transmitted from the robot controller. System integration was tested in a bin picking application, where objects were picked from a cluttered bin and placed in another container. The system was able autonomously and reliably perform the specified tasks, demonstrating the applicability of the system to industrial component handling tasks.

9.2 Further Work

To further improve the performance of the implemented pose estimation system, some shortcomings should be addressed. The occurrence of false positive matches can be reduced, or eliminated outright, by implementing the per object score limit, in accordance with the discussion in section 8.2. Since the processing time is closely linked to the currently used point pair feature method, it is unlikely that this can be improved. If faster pose estimation is required, a different method for pose estimation should be implemented as a new module of the pose estimation system.

The fallback effort in the retry system of the linear Cartesian MoveIt 2 motion planning pipeline should be altered. The motions performed to find a different manipulator configuration should be simulated, and not executed on the robot. This can be done by extracting the last RobotState of a found trajectory and imposing it on the PlanningComponent. This should allow planning to be performed from the configuration defined in the imposed RobotState. After two motions have been simulated, a linear Cartesian planning attempt can be performed from the resulting RobotState. Additionally, the equivalent state effort can be altered to more reliably find configurations yielding the same end-effector pose. This can be achieved by extending the predefined list of seeds or using an analytical inverse kinematics solver.

The paths found by RRT-Connect are rarely direct and often involve unnecessary motions. The newest version of the MoveIt 2 Beta include functionality to change the employed motion planner. To produce more direct paths, a sampling-based planner optimizing for short paths can be selected. Specifically, the potential of the RRT* algorithm should be investigated.

Currently the trajectory generated by the path planner is used directly to control the manipulator. This can in some cases result in jagged motions. If smoother manipulator motion is desired, joint trajectory interpolation capabilities should be implemented as part of the trajectory controller.

References

[1] *ABB unveils the future of human-robot collaboration: YuMi®*. 2014. URL: https://new.abb.com/news/detail/13110/abb-unveils-the-future-of-human-robot-collaboration-yumir (visited on 03/30/2020).

[2] *abb_libegm*. URL: https://github.com/ros-industrial/abb_libegm (visited on 05/22/2020).

[3] *abb_librws*. URL: https://github.com/ros-industrial/abb_librws (visited on 05/25/2020).

[4] *About ROS*. URL: https://www.ros.org/about-ros/ (visited on 03/29/2020).

[5] *Announcing MoveIt 1.0 and a Master Branch*. 2019. URL: https://moveit.ros.org/moveit!/ros/2019/03/08/announcing-the-moveit-1-release.html (visited on 03/27/2020).

[6] *Application manual - Controller software IRC5*. Rev. C. ABB Robotics. 2016.

[7] Zivid AS. *Zivid SDK*. 2020. URL: http://www.zivid.com/sdk (visited on 05/22/2020).

[8] Paul J Besl and Neil D McKay. “Method for registration of 3-D shapes”. In: *Sensor fusion IV: control paradigms and data structures*. Vol. 1611. International Society for Optics and Photonics. 1992, pp. 586–606.

[9] Markus Bjønnes. “Robotic Hand Eye Calibration using Point Clouds”. Delivery in the specialization project, in the course TPK4560. Dec. 2019.

[10] Eric Brachmann, Alexander Krull, Frank Michel, Stefan Gumhold, Jamie Shotton, and Carsten Rother. “Learning 6d object pose estimation using 3d object coordinates”. In: *European conference on computer vision*. Springer. 2014, pp. 536–551.

[11] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. “Geometric deep learning: going beyond euclidean data”. In: *IEEE Signal Processing Magazine* 34.4 (2017), pp. 18–42.

[12] Sachin Chitta, Ioan Sucan, and Steve Cousins. “Moveit![ros topics]”. In: *IEEE Robotics & Automation Magazine* 19.1 (2012), pp. 18–19.

[13] Peter Corke. *Robotics, vision and control: fundamental algorithms in MATLAB® second, completely revised*. Vol. 118. Springer, 2017.

[14] Alessandro De Luca, Alin Albu-Schaffer, Sami Haddadin, and Gerd Hirzinger. "Collision detection and safe reaction with the DLR-III lightweight manipulator arm". In: *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2006, pp. 1623–1630.

[15] Alessandro De Luca and Raffaella Mattone. "Sensorless robot collision detection and hybrid force/motion control". In: *Proceedings of the 2005 IEEE international conference on robotics and automation*. IEEE. 2005, pp. 999–1004.

[16] Bertram Drost and Slobodan Ilic. "2012 Second International Conference on 3D Imaging, Modeling, Processing, Visualization & Transmission". In: IEEE. 2012, pp. 9–16.

[17] Bertram Drost, Markus Ulrich, Nassir Navab, and Slobodan Ilic. "Model globally, match locally: Efficient and robust 3D object recognition". In: *2010 IEEE computer society conference on computer vision and pattern recognition*. Ieee. 2010, pp. 998–1005.

[18] Jonatan S Dyrstad, Marianne Bakken, Esten I Grøtli, Helene Schulerud, and John Reidar Mathiassen. "Bin Picking of Reflective Steel Parts using a Dual-Resolution Convolutional Neural Network Trained in a Simulated Environment". In: *2018 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. IEEE. 2018, pp. 530–537.

[19] Olav Egeland. *Robot Vision*. Jan. 2019.

[20] Hk ElMaraghy and W ElMaraghy. "Smart adaptable assembly systems". In: *Procedia CIRP* 44.4-13 (2016), pp. 127–128.

[21] Asa Fasth, Johan Stahre, and Kerstin Dencker. "Level of automation analysis in manufacturing systems". In: *Advances in Human Factors, Ergonomics, and Safety in Manufacturing and Service Industries. Chapter* (2010), pp. 233–242.

[22] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Irvine, 2000.

[23] *FlexPendant Development*. 2016. URL: http://developercenter.robotstudio.com/flexpendant/manuals (visited on 04/01/2020).

[24] Andreas Georgopoulos, Ch Ioannidis, and A Valanis. "Assessing the performance of a structured light scanner". In: *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences* 38.Part 5 (2010), pp. 251–255.

[25] *HALCON Operator Reference, 3D Matching.* URL: https://www.mvtec.com/doc/halcon/13/en/find_surface_model.html (visited on 06/08/2020).

[26] Christopher G Harris, Mike Stephens, et al. “A combined corner and edge detector.” In: *Alvey vision conference.* Vol. 15. 50. Citeseer. 1988, pp. 10–5244.

[27] Richard S Hartenberg and Jacques Denavit. “A kinematic notation for lower pair mechanisms based on matrices”. In: *Journal of applied mechanics* 77.2 (1955), pp. 215–221.

[28] Stefan Hinterstoisser, Stefan Holzer, Cedric Cagniart, Slobodan Ilic, Kurt Konolige, Nassir Navab, and Vincent Lepetit. “Multimodal templates for real-time detection of texture-less objects in heavily cluttered scenes”. In: *2011 international conference on computer vision.* IEEE. 2011, pp. 858–865.

[29] Stefan Hinterstoisser, Vincent Lepetit, Slobodan Ilic, Stefan Holzer, Gary Bradski, Kurt Konolige, and Nassir Navab. “Model based training, detection and pose estimation of texture-less 3d objects in heavily cluttered scenes”. In: *Asian conference on computer vision.* Springer. 2012, pp. 548–562.

[30] Tomáš Hodaň, Jiri Matas, and Stepan Obdrzalek. “On evaluation of 6D object pose estimation”. In: *European Conference on Computer Vision.* Springer. 2016, pp. 606–619.

[31] Tomáš Hodaň, Frank Michel, Eric Brachmann, Wadim Kehl, Anders Glent-Buch, Dirk Kraft, Bertram Drost, Joel Vidal, Stephan Ihrke, Xenophon Zabulis, et al. “Bop: Benchmark for 6d object pose estimation”. In: *Proceedings of the European Conference on Computer Vision (ECCV).* 2018, pp. 19–34.

[32] Tomáš Hodaň, Xenophon Zabulis, Manolis Lourakis, Štěpán Obdržálek, and Jiřı Matas. “Detection and fine 3D pose estimation of texture-less objects in RGB-D images”. In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* IEEE. 2015, pp. 4421–4428.

[33] Radu Horaud and Fadi Dornaika. “Hand-eye calibration”. In: *The international journal of robotics research* 14.3 (1995), pp. 195–210.

[34] Svenja Kahn, Dominik Haumann, and Volker Willert. “Hand-eye calibration with a depth camera: 2D or 3D?” In: *2014 International Conference on Computer Vision Theory and Applications (VISAPP).* Vol. 3. IEEE. 2014, pp. 481–489.

[35] James J Kuffner and Steven M LaValle. “RRT-connect: An efficient approach to single-query path planning”. In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065).* Vol. 2. IEEE. 2000, pp. 995–1001.

[36] Tobias Kunz and Mike Stilman. “Time-optimal trajectory generation for path following with bounded acceleration and velocity”. In: *Robotics: Science and Systems VIII* (2012), pp. 1–8.

[37] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[38] Yi Li, Gu Wang, Xiangyang Ji, Yu Xiang, and Dieter Fox. “DeepIM: Deep Iterative Matching for 6D Pose Estimation”. In: *The European Conference on Computer Vision (ECCV)*. Sept. 2018.

[39] Kevin M Lynch and Frank C Park. *Modern Robotics*. Cambridge University Press, 2017.

[40] *Managed nodes*. URL: https://design.ros2.org/articles/node_lifecycle.html (visited on 06/08/2020).

[41] Farzin Mokhtarian and Riku Suomela. “Robust image corner detection through curvature scale space”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20.12 (1998), pp. 1376–1381.

[42] *Motion Planning Pipeline — moveit_tutorials Melodic documentation*. URL: https://ros-planning.github.io/moveit_tutorials/doc/motion_planning_pipeline/motion_planning_pipeline_tutorial.html (visited on 03/27/2020).

[43] *MoveItCpp Tutorial — moveit_tutorials Melodic documentation*. URL: https://ros-planning.github.io/moveit_tutorials/doc/moveit_cpp/moveitcpp_tutorial.html (visited on 03/27/2020).

[44] *MVTec HALCON*. URL: https://www.mvtec.com/products/halcon/ (visited on 06/07/2020).

[45] Davide Nicolis, Andrea Maria Zanchettin, and Paolo Rocco. “Constraint-based and sensorless force control with an application to a lightweight dual-arm robot”. In: *IEEE Robotics and Automation Letters* 1.1 (2016), pp. 340–347.

[46] Marius Nilsen. “ROS2 Integration of ABB IRB 14000 YuMi”. Delivery in the specialization project, in the course TPK4560. Dec. 2019.

[47] *Open Multi-Processing*. URL: https://www.openmp.org/.

[48] *Operating Manual - IRB 1400*. Rev. E. ABB Robotics. 2018.

[49] *Orocos Kinematics and Dynamics*. URL: https://www.orocos.org/kdl (visited on 06/07/2020).

[50] Jia Pan, Sachin Chitta, and Dinesh Manocha. “FCL: A general purpose library for collision and proximity queries”. In: *2012 IEEE International Conference on Robotics and Automation*. IEEE. 2012, pp. 3859–3866.

[51] Frank C Park and Bryan J Martin. "Robot sensor calibration: solving AX = XB on the Euclidean group". In: *IEEE Transactions on Robotics and Automation* 10.5 (1994), pp. 717–721.

[52] *Point Cloud Library*. URL: https://pointclouds.org/.

[53] Matteo Parigi Polverini, Andrea Maria Zanchettin, Sebastiano Castello, and Paolo Rocco. "Sensorless and constraint based peg-in-hole task execution with a dual-arm robot". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2016, pp. 415–420.

[54] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. "Pointnet: Deep learning on point sets for 3d classification and segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 652–660.

[55] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. "Faster r-cnn: Towards real-time object detection with region proposal networks". In: *Advances in neural information processing systems*. 2015, pp. 91–99.

[56] *Robot Web Services*. URL: http://developercenter.robotstudio.com/blobproxy/devcenter/Robot_Web_Services/html/index.html (visited on 04/05/2020).

[57] *ros2_control*. URL: https://github.com/ros-controls/ros2_control (visited on 05/21/2020).

[58] *rviz_wiki*. URL: https://github.com/ros2/rviz (visited on 05/25/2020).

[59] Yiu Cheung Shiu and Shaheen Ahmad. "Calibration of wrist-mounted robotic sensors by solving homogeneous transform equations of the form AX = XB". In: *ieee Transactions on Robotics and Automation* 5.1 (1989), pp. 16–29.

[60] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: modelling, planning and control*. Springer Science & Business Media, 2010.

[61] Stephen M Smith and J Michael Brady. "SUSAN—a new approach to low level image processing". In: *International journal of computer vision* 23.1 (1997), pp. 45–78.

[62] *StateMachine Add-In 1.1*. URL: https://robotapps.robotstudio.com/#/viewApp/c163de01-792e-4892-a290-37dbe050b6e1 (visited on 05/20/2020).

[63] Ioan A Sucan, Mark Moll, and Lydia E Kavraki. "The open motion planning library". In: *IEEE Robotics & Automation Magazine* 19.4 (2012), pp. 72–82.

[64] Roger Tsai. "A versatile camera calibration technique for high-accuracy 3D machine vision metrology using off-the-shelf TV cameras and lenses". In: *IEEE Journal on Robotics and Automation* 3.4 (1987), pp. 323–344.

[65] Roger Y Tsai and Reimar K Lenz. “A new technique for fully autonomous and efficient 3D robotics hand/eye calibration”. In: *IEEE Transactions on robotics and automation* 5.3 (1989), pp. 345–358.

[66] *User Manual StateMachine Add-In 1.1*. ABB AB, Robotics. 2019.

[67] Joel Vidal, Chyi-Yeu Lin, and Robert Martı. “6D pose estimation using an improved method based on point pair features”. In: *2018 4th international conference on control, automation and robotics (iccar)*. IEEE. 2018, pp. 405–409.

[68] Chen Wang, Danfei Xu, Yuke Zhu, Roberto Martın-Martın, Cewu Lu, Li Fei-Fei, and Silvio Savarese. “Densefusion: 6d object pose estimation by iterative dense fusion”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 3343–3352.

[69] *What is DDS?* URL: https://www.dds-foundation.org/what-is-dds-3/ (visited on 04/03/2020).

[70] Yu Xiang, Tanner Schmidt, Venkatraman Narayanan, and Dieter Fox. “Posecnn: A convolutional neural network for 6d object pose estimation in cluttered scenes”. In: *arXiv preprint arXiv:1711.00199* (2017).

[71] *YuMi Data Sheet*. Rev. H. ABB Robotics. 2019.

[72] Menglong Zhu, Konstantinos G Derpanis, Yinfei Yang, Samarth Brahmbhatt, Mabel Zhang, Cody Phillips, Matthieu Lecce, and Kostas Daniilidis. “Single image 3D object detection and pose estimation for grasping”. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2014, pp. 3936–3943.

Markus Bjønnes, Marius Nilsen

NTNU

Norwegian University of
Science and Technology