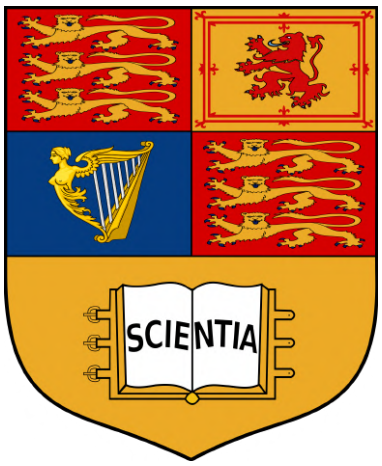
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2022



Project Title:	**Dynamic Grasping with a Quadruped Mobile Manipulator**
Student:	**Arman Fidanoglu**
CID:	**01512561**
Course:	**EEE4**
Project Supervisor:	**Prof Yiannis Demiris**
Second Marker:	**Dr Ad Spiers**

Final Report Plagiarism Statement

I affirm that I have submitted, or will submit, an electronic copy of my final year project report to the provided EEE link.

I affirm that I have submitted, or will submit, an identical electronic copy of my final year project to the provided Blackboard module for Plagiarism checking.

I affirm that I have provided explicit references for all the material in my Final Report that is not authored by me, but is represented as my own work.

Acknowledgements

I would like to thank Professor Yiannis Demiris, my supervisor for this project, for presenting me with the opportunity to be a part of the cutting edge in robotics and allowing me to shape the path of this unique project.

I would also like to thank Rodrigo and Cedric from Imperial's Personal Robotics Laboratory, who continuously supported me throughout the project. Without your patience and help, this project would not have come to life.

Without doubt I would like to thank my friends Spyros, Vasilis and Udai, for sharing the stress and making the library sessions infinitely more enjoyable. The project was enhanced by your presence.

Above all, I thank my family for always being by my side - they are the sole reason I am in a position to write this report today, and I could never repay them for any of the sacrifices they have made. Thank you for everything... Sizi seviyorum!

Abstract

The adoption of mobile manipulators encompasses more industries every day. Although their applications are rapidly increasing, performing an autonomous grasp with a mobile manipulator is still slow, as the movement of the base and arm is a sequential motion.

Enabling a base and robotic arm to move simultaneously would significantly speed up mobile manipulation. This project utilises computer vision techniques to detect an object and localise it in three-dimensional space. It then integrates these with robotics techniques to perform an autonomous "dynamic" grasp with a mobile manipulator, where the base and robotic arm move concurrently.

The outcome of the project is a complete dynamic grasping pipeline, which acts as a proof of concept to pave the way of future research in mobile manipulation. The conducted tests show the potential of this system, along with its challenges. This report discusses the achievements and shortcomings of the developed product.

Contents

List of Figures

List of Tables

List of Source Code Listings

Chapter 1

Introduction

1.1 Motivation and Objectives

Mobile manipulators are a subject of focus in research and development, owing to the very wide range of applications for autonomous robots that can interact with objects around them. Such applications encompass several industries (only some of which have been listed below), and can be grouped into two categories, based on terrain type:

1. Rough and unpredictable terrain: Exoplanet exploration, reconnaissance, Search and rescue operations, agriculture, mining, construction, etc.

2. Flat and predictable terrain: Home care, cleaning, healthcare, manufacturing, industrial inspection, etc.

In both categories, there has been extensive research into wheeled and tracked robots [Semini et al., 2011]. Wheeled robots have been used as extensively as they have, mainly because they are easy to control, stable, cost-effective and energy efficient. Indeed, legged robots need considerable energy just to keep their bodies off the floor [Heppner et al., 2014].

There might not be a need to use legged (quadruped and biped) robots in indoor applications that do not require climbing stairs, such as healthcare or home care (single floor), as these are much more costly and wheeled robots perform well on flat surfaces. However, they prove very useful in the first category (rough terrain) and on multiple floors, as they have advantages in terms of mobility and versatility [Hutter et al., 2016], and they can outperform both wheeled and tracked robots on rough terrain [Semini et al., 2011].

In recent decades, there has been extensive research into legged robots, that has led to the developments of quadruped robots such as the HyQ [Semini et al., 2011], ANYmal [Hutter et al., 2016], ALMA [Bellicoso et al., 2019], to cite a few. However, there has not been as much research into their use as a mobile manipulator [Heppner et al., 2014]. The only piece of research about legged mobile manipulators that the author was able to find is a paper that describes the optimal configuration (position, power input, communication protocols) of a manipulator on a quadruped robot [Rehman et al., 2016].

Quadruped mobile manipulators exist, but autonomous grasping is currently done sequentially, where the robot first moves to a suitable position, followed by the manipulator arm performing an autonomous grasp. There is now a need to build on this research and to use this optimal configuration to develop software for a better quadruped mobile manipulator. Such a mobile manipulator would need to grasp objects much faster than the current state-of-the-art robots.

Enabling the robot and arm to move simultaneously to grasp an object in one, continuous motion would be a significant step in achieving this goal. To this end, below are the objective and motivation for this project.

Objective: To develop perception and control algorithms to enable simultaneous movement of the two components of a quadruped mobile manipulator, robot body and arm, when performing a moving (dynamic) grasp.

Reason: To speed up grasping with quadruped mobile manipulators, allow for a more natural motion when grasping and to aid future research involving mobile manipulators.

1.2 Requirements Capture

The software developed as part of this project aims to implement a complete grasping pipeline involving both the quadruped robot and the manipulator arm. The software must be able to:

1. Use information from visual and depth cameras in order to detect and localise objects in three-dimensional space,
2. Grasp objects while stationary,
3. Create a trajectory for the quadruped robot to the detected object's proximity,
4. Create a motion plan for the manipulator to reach the object when the quadruped robot reaches its final position,
5. Coordinate the two movements such that the arm has reached the object by the time the quadruped robot has stopped moving,
6. Successfully perform a grasp and pick up the object, in a shorter time than would be necessary for sequential movement and grasping.

Contributions to meet requirements

In order to meet the requirements, the project aims to contribute software which can recognise an object from a distance, localise it (Calculate real-world coordinates), move towards it and perform a complete grasp. In addition, it aims to speed up the grasping process by integrating the grasp and movement motions into one continuous motion ("dynamic grasping"). The contributions will be detailed in Chapter 3.

If successful, achieving this goal will improve the efficiency of and shorten time required for testing in future projects which require grasping with a mobile manipulator, as well as industrial applications for mobile manipulators.

1.3 Project Specification and Scope

The list below details the hardware used in the project, along with the motivations for using each component.

- BostonDynamics Spot quadruped robot, chosen for its comprehensive API, effective movement capabilities and effectiveness in carrying payloads,

- Kinova Gen3 manipulator arm with 7 degrees of freedom, chosen for being robust, light and portable,

- NVIDIA Xavier NX Developer kit (For object recognition and image processing), chosen for its low power consumption and high image processing performance.

The above setup is demonstrated in Figure 1.1.

Figure 1.1: Spot and Kinova: Configuration of system components

This project involves detecting and grasping an object from a large distance, but not navigation and exploration to find and differentiate between objects to be picked up. This means that the object to be picked up must be in sight of the robot, along with other necessary conditions. Therefore, it is necessary to define the project's scope clearly.

In order to define the project scope, assumptions about the environment and starting configuration were made. These assumptions are as follows.

1. The object to be grasped is in the direct line of sight of the camera module on the Kinova arm,

2. There may be multiple objects in the scene, but only one kind of test object will be considered for grasping,

3. Spot is close enough to the object to recognise it through the camera module on the Kinova arm but far enough that the manipulator cannot reach the object even when fully extended,

4. The object is on the same flat plane (floor) as Spot and there are no obstacles in the way.

An additional note is that for item 2 in the list above, a teddy bear was chosen due to its distinct features.

In addition to the assumptions, the project objective must be specified in greater detail to rigidify the scope of the project. Performing a distant grasp in one motion would entail developing a feedback system to continuously correct the robot's motions during the movement, such that the gripper ends up precisely on the object by the time the quadruped robot has stopped moving. Due to time constraints, this project will not attempt to develop the required feedback systems to correct the motion of the robot and arm during movement.

Instead, the dynamic grasp will be broken into two stages. During the first stage, the arm will move close to the object as the robot moves into grasping distance of the object. At the end of the first stage, the robot will be in an intermediate state such that the arm is hovering right over the object. In the second stage, a second arm movement will be attempted, starting from the intermediate state. This movement will end in the required grasp.

Although this process is not one continuous motion, it reduces the time required to perform a grasp from a distance. This is thanks to the second grasp stage starting from the intermediate state, rather than the home state, which is a shorter motion.

1.4 Report Structure

The body of report consists of five sections:

1. Background and Prior Work
2. Analysis and Design
3. Implementation
4. Testing and Results
5. Discussions: Evaluation, Conclusions and Further Work

This structure guides the reader through the project's lifecycle, starting with the research in the field and the context in which the project is situated. This information is then used to explain the conceptual design process, implementation and testing stages. Finally, all software created and results obtained are evaluated in the last section, ending with the future work that the project might lead to.

Chapter 2

Background Theory and Prior Work

There are multiple technologies involved in performing a continuous grasp. To grasp an object, the robot first needs to detect it. Then it must localise the object and plan a trajectory for the base and arm, such that the gripper ends up hovering over the object. Finally, it must plan a new trajectory for the arm, to grasp the object. The robot must also execute the two motion plans simultaneously and to end up at the desired position.

This chapter explores prior work in each of the above stages individually, as each stage is crucial to the success of the project. For each stage, the relevant aspects of the background theory will be introduced, followed by the research results and comments.

2.1 Object Detection and Localisation

Object detection is a computer vision technique for locating instances of objects in images or videos. The technique is important for this project, since objects must be detected in order to perform an autonomous grasp.

2.1.1 What Aspects Were Researched?

Since dynamic manipulation has potential uses across a variety of industries and applications, the system must ultimately recognise a large variety of objects and accurately locate them. Furthermore, the detection must happen in real time, as the system must detect and localise objects as it moves around in its surroundings.

Therefore, the focus will be on general object detection, with a large number of object classes. Additionally, we will research object detection models that are small or efficient enough to run in real-time, and compare their performances.

2.1.2 The Methods

Modern object detection methods can be divided into one- and two-stage approaches. One-stage detectors are more efficient owing to straightforward architectures, but the two-stage detectors still take the lead in accuracy [Lu et al., 2020]. Since this project requires real-time object detection, efficiency is preferred over accuracy. Therefore, only one-stage detectors will be explored.

SSD: Single Shot MultiBox Detector

SSD detects objects in images using a single deep neural network. In order to speed up detection, it discretizes the output space of bounding boxes into a set of default boxes over different aspect ratios and scales per feature map location. At prediction time, the network produces adjustments to the box to better match the object shape. This makes SSD easy to train and straightforward to integrate into systems that require a detection component. Experimental results on the PASCAL VOC, COCO, and ILSVRC datasets confirm that SSD has competitive accuracy to two-stage detectors and is much faster [Liu et al., 2016]. To the best of the knowledge of the paper authors, SSD300 is the first real-time method to achieve above 70% accuracy.

MobileNet architecture

MobileNets are a class of efficient neural network models created for mobile and embedded vision applications. MobileNets are based on a streamlined architecture that uses depthwise separable convolutions to build **lightweight** deep neural networks. This is done by introducing two simple global hyperparameters that efficiently trade off between latency and accuracy [Howard et al., 2017].

SSD-MobileNet-v2

The SSD Detector was initially based on the computationally expensive VGG16 [Simonyan and Zisserman, 2014] architecture. To reduce the computational complexity of the VGG16-SSD detector, researchers at Google replaced the VGG16 network with the MobileNet architecture, improving the real-time performance of the SSD detector. The authors of [Chiu et al., 2020] have now based the SSD detector on the improved MobileNet-v2 [Sandler et al., 2018] model. Experimental results show that the proposed MobileNet-SSD-v2 detector not only retains the advantage of fast processing of the original MobileNet-SSD detector, but also greatly improves the detection accuracy.

Other Models

YOLO [Redmon et al., 2015], much like SSD, is one of the most popular one-stage object detectors. After them, RetinaNet [Lin et al., 2017] and M2det [Zhao et al., 2018] took over with better accuracy. However, in order to improve accuracy rate, these new object detectors use extremely high computational backbone networks such as VGG16 or ResNet [He et al., 2015] in the design of feature extraction networks, similar to VGG16-SSD. Moreover, the YOLO detector is not tailored to detect dense multiple targets and large objects. Instead, the SSD detector is more suitable for detecting a large number of objects with different scales [Chiu et al., 2020].

2.1.3 Comments

Based on the information gathered about various object detection models, SSD-MobileNet-v2 stands out for a few reasons, as listed below.

- As a combination of lightweight and accurate models, it performs with high accuracy, with latency low enough to be suitable for real-time applications,
- It is much more lightweight than both two-stage object detectors and other state-of-the-art one-stage detectors,
- It is easy to re-train, which is useful for the project since the system will be trained to detect a test object accurately.

2.2 Object Pose Estimation and Grasp Generation

In order to plan a trajectory for a robotic grasp, a robot must know the end-effector pose that its manipulator arm will move to. To generate an end-effector position that will end up in a successful grasp, grasp pose selection is performed.

Grasp pose selection is one of the most important problems in robot manipulation. Here, a robot observes an object and needs to decide where to move its gripper (3D position and 3D orientation) in order to pick up the object. Grasp selection is highly complex, since the stability of grasps depends on object and gripper geometry, object mass distribution, and surface frictions [Mousavian et al., 2019]. There are various considerations which affect the quality of the grasp generated by the grasp generator: These are explored in the following sections.

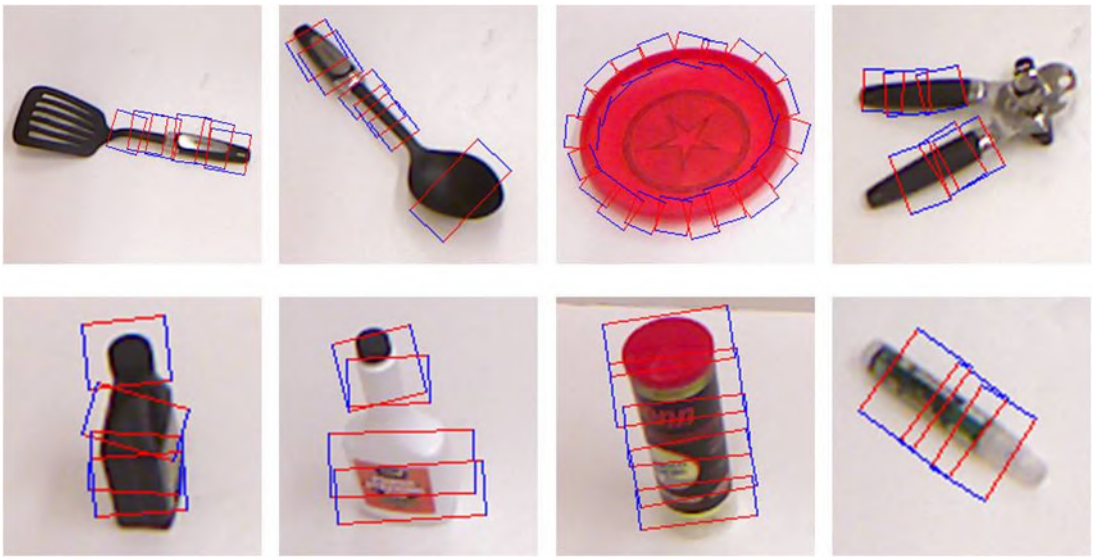


Figure 2.1: Sets of grasp rectangles on Cornell Grasp Dataset objects, generated by GR-ConvNet

2.2.1 What Aspects Were Researched?

Grasp generators such as MoveIt Grasps [MoveIt!, b] require a 6-dimensional object pose (the object's orientation in three-dimensional space, in addition to its position). Such grasp generators generate a large number of potential grasp approaches and directions, which also have the constraint of being kinematically feasible. Obtaining an object pose is important for robotic grasping, as the robot must possess knowledge of the best gripper orientations to grasp the object successfully. Therefore, Various methods of estimating object poses were explored.

2.2.2 The Methods

Object pose estimation via deep learning

There exist several deep neural network models aiming to estimate object poses and therefore a set of grasp poses, from a single RGB-D pointcloud image ([Kleeberger et al., 2021], [Mousavian et al., 2019], [Gualtieri et al., 2016], [Kumra et al., 2019]). Out of all of the above networks, GR-ConvNet was found to have achieve the highest accuracy on the Cornell Grasping Dataset (CGD), which has common household items. Example grasp poses generated on an RGB-D image are shown in Figure 2.1.

Grasp generation without object pose estimation

There are several cases where objects that require grasping are "irregular", meaning that the object cannot be grasped from all angles. An example to this could be a

pencil, spatula, or any other elongated object, which should be grasped by its handle or body, but cannot be grasped by placing the grippers on either end.

However, there are also cases where objects can be grasped in any orientation, such as a teddy bear or a tennis ball. In these cases, the manipulator arm can grasp the object from any angle as long as the generated end-effector position is kinematically feasible.

For use in such cases, researchers in the Personal Robotics Laboratory have come across a grasp generator which uses spherical object approximations [Pal-Robotics], which was initially used for an object pick-and-place demo for the TIAGo mobile manipulator robot [Pagès et al., 2016].

2.2.3 Comments

Grasp pose generation is an important step in the pipeline. However, using a deep neural network-based object pose estimator is computationally expensive and might have a negative impact on the real-time performance of the system. On the other hand, although the spherical object approximations work only with a niche class of objects, this is deemed acceptable for the purposes of this project, since the focus is on robotic grasping rather than computer vision.

2.3 Motion Planning

To generate a grasping motion, an inverse kinematics problem must be solved. Inverse Kinematics (IK) is defined as the problem of determining a set of appropriate joint configurations for which the end-effector moves to desired positions as smoothly, rapidly, and as accurately as possible [Aristidou and Lasenby, 2009]. The desired position is the grasp pose determined by the grasp pose generator.

2.3.1 What Aspects Were Researched?

This is a field that has been thoroughly explored, with countless papers describing various ways of solving the IK problem. This research has resulted in the development of off-the-shelf IK solvers, such as TRAC-IK [Beeson and Ames, 2015]. Additionally, there are many motion planners to choose from, which already incorporate inverse kinematics solvers. Therefore, the research is concentrated around which motion planning library performs the best given the constraints of the project.

2.3.2 The Methods

There are a plethora of grasp generators and motion planners to use for this task, such as:

- OMPL (Open Motion Planning Library) [Şucan et al., 2012],
- Pilz Industrial Motion Planner [MoveIt!, d],
- Stochastic Trajectory Optimization for Motion Planning (STOMP) [Kalakrishnan et al., 2011],
- Search-Based Planning Library (SBPL) (Library containing various generic planners) [SBPL],
- Covariant Hamiltonian Optimization for Motion Planning (CHOMP) [Ratliff et al., 2009].

All of the above planners have been used with MoveIt [MoveIt!, c], which is the library that was chosen for this task. OMPL, CHOMP and STOMP have been quoted by MoveIt to create the smoothest motions. Therefore, these three motion planners were compared to choose a motion planner.

OMPL vs CHOMP vs STOMP

OMPL is an open source library for sampling based/randomized motion planning algorithms. Sampling based algorithms are probabilistically complete: A solution would eventually be found if one exists, however non-existence of a solution cannot be reported. These algorithms are efficient and usually find a solution quickly [MoveIt!, e].

[MoveIt!, e] also provides a qualitative analysis is performed for these planners. A short summary is provided in four bullet points.

- STOMP and OMPL can avoid local minima due to their stochastic nature, whereas CHOMP gets stuck in local minima, avoiding the optimal solution.
- Although execution times are comparable, OMPL algorithms are efficient and take fewer iterations than CHOMP and STOMP.
- CHOMP generally requires additional parameter tuning than STOMP to obtain a successful solution, while OMPL usually performs well with the default parameters.

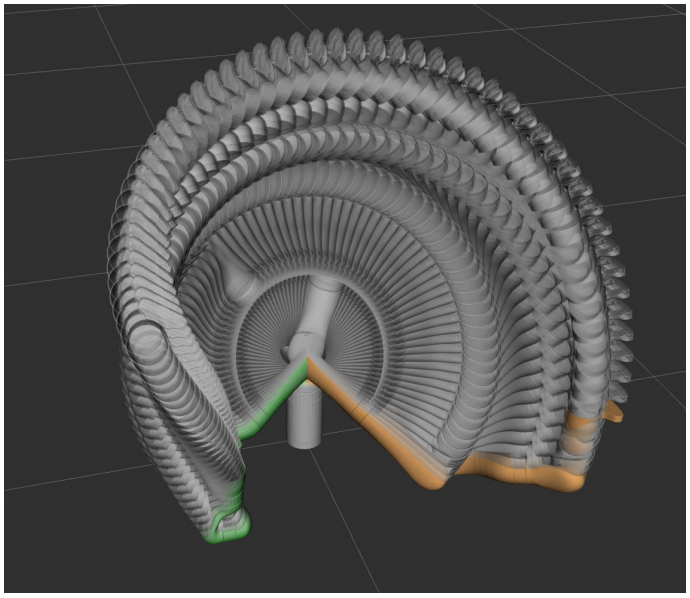


Figure 2.2: Example of a trajectory planned with OMPL for the Kinova Gen3 Manipulator, visualised in RViz

- STOMP and OMPL generate smooth and collision-free trajectories, whereas CHOMP generates paths which do not prefer smooth trajectories.

On top of the above analysis, OMPL is also the motion planner most compatible with and best integrated into MoveIt!.

2.3.3 Comments

As part of the MoveIt Motion Planning Framework, there are several motion planners to choose from. However, due to its efficiency in finding a solution, the smoothness and quality of its solutions and compatibility with MoveIt!, OMPL is preferred for the project. Figure 2.2 demonstrates these properties in a motion plan generated with OMPL.

2.4 Mobile Manipulation

The problem of motion planning for *mobile manipulators*, or solving for a path for a manipulator on a moving base, is an entirely different field than stationary motion planning. Several mobile manipulator motion planning were explored for the project. The research will be explained in this section.

2.4.1 What Aspects Were Researched?

There are three categories of mobile manipulators: wheeled, tracked and legged. The field of wheeled and tracked mobile manipulators have been well studied and different types of models, depending on the application, have been developed in the recent decade [Bhavanibhatla and Pratihar, 2017]. On the other hand, inverse kinematics for legged mobile manipulators is relatively unexplored, with one exception being [Bhavanibhatla and Pratihar, 2017], which describes an IK solution for a rectangular hexapod robot.

It is necessary to look into all three of these fields (IK for wheeled, tracked and legged mobile manipulators) to determine if an appropriate solution exists for this project.

2.4.2 The Methods

IK for wheeled and tracked mobile manipulators (WMM and TMM)

Several papers look into IK for wheeled and tracked mobile manipulators. These include [Thakar et al., 2022] and [Bayle et al., 2003] for WMM, as well as [Liu and Liu, 2009b], [Liu and Liu, 2007], [Liu and Liu, 2009c] and [Liu and Liu, 2009a] for TMM. These papers describe a method of modelling an optimal inverse kinematics solution to a manipulator mounted on a wheeled (tracked) robot. These two categories were grouped together, as they both behave differently to legged mobile manipulators.

A scenario commonly solved for in these papers is allowing the manipulator to pick up an object on the move, while the robot is moving on a continuous, non-straight path. The calculations take into account the rotation of the axle of the wheels [Bayle et al., 2003] or track slippage [Liu and Liu, 2009c]. However, a legged robot will turn in a completely different way while picking up the object, which makes all of the research in these fields inapplicable to this project. Due to these differences, solutions belonging to this category will not be considered for this project.

IK for legged mobile manipulators (LMM)

The hexapod robot in [Bhavanibhatla and Pratihar, 2017] will also move in a different way to Spot, which is a quadruped robot. Additionally, since some of the technology in Spot is a trade secret, and to minimise user error, Spot's individual-limb controls remain inaccessible to the user [Zimmermann et al., 2021]. This makes it impossible to use the solution in [Bhavanibhatla and Pratihar, 2017] for the purposes of this project.

Due to the issues mentioned in both categories above, a mobile manipulator IK solution will not be used for this project.

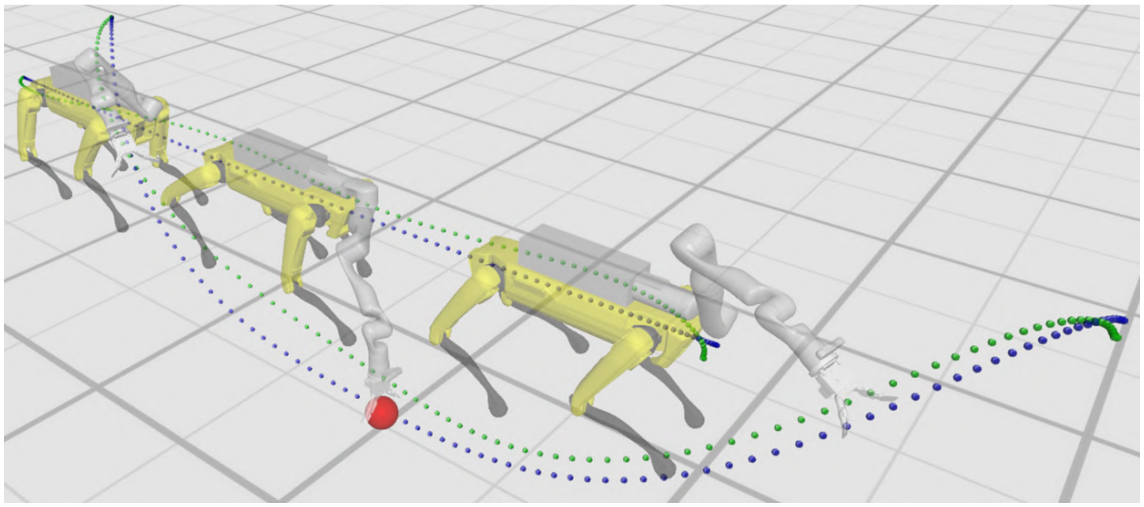


Figure 2.3: Spot snatching a ball in a continuous motion, optimal route to counter varying manipulator forces. From [Zimmermann et al., 2021]

Stationary manipulator motion planning with position corrections

Researchers in ETH Zurich created a project named "Go Fetch!" wherein the authors' objective was to create a control algorithm to perform dynamic grasps with Spot and the Kinova manipulator. The premise of this project is that it treats Spot as a black box, since its technology is proprietary and users do not have access to individual limbs [Zimmermann et al., 2021].

"Go Fetch!" focuses on dealing with the forces induced on Spot by the arm when it grasps objects in a sweeping motion while it walks by, as demonstrated in Figure 2.3.

Go Fetch! accomplishes a similar goal to this project, which is to perform a dynamic grasp. However, the project does not aim to perform an autonomous dynamic grasp, but rather creating an optimisation algorithm to counter the forces induced by the independently-moving Kinova arm when grasping an object at a known position.

2.4.3 Comments

As per the research conducted in this section, a mobile manipulator motion planning solution was not selected, due to the unavailability of an efficient solution to achieve the project's objective. Additionally, Spot's proprietary nature does not allow access to individual limbs, which would render such a solution infeasible.

Instead, the approach will be similar to *Go Fetch!*, where a stationary motion planner is used in conjunction with various algorithms to achieve the goal. This project will differ from *Go Fetch!* in that rather than snatching an object from a known location, an autonomous dynamic grasp will be attempted.

Chapter 3

Analysis and Design

In this section, an overview of the system design is provided and explored. This includes the final design and the choices made to get there, as well as design considerations. The project explored various possible designs and methods, as well as components, to achieve the required motion. The changes to the design, scope and goals throughout the process will also be explored here.

3.1 Design Overview

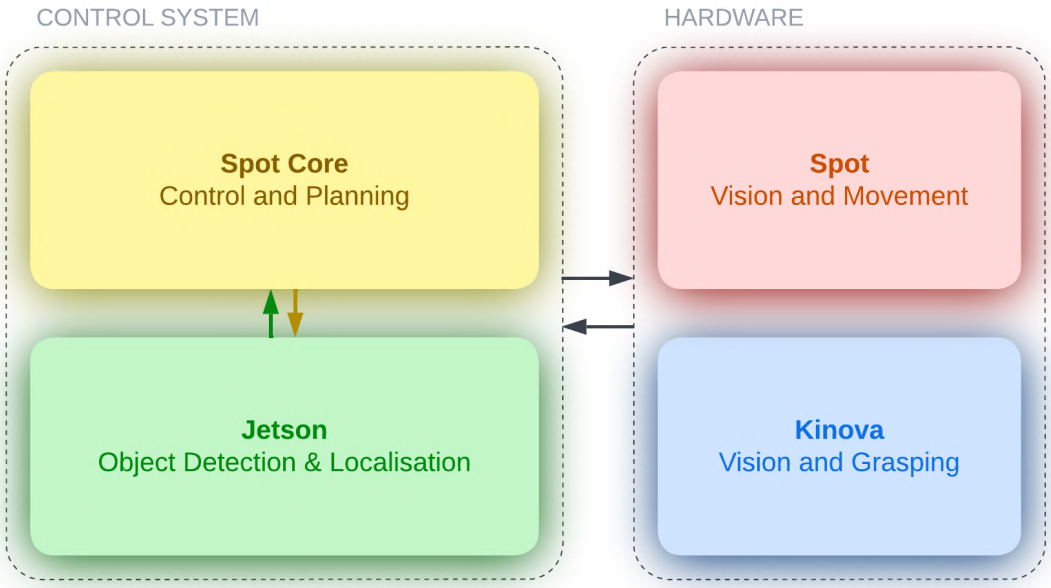


Figure 3.1: High-level overview of system design

The system comprises four components:

1. The BostonDynamics Spot robot (Will be referred to as **Spot**) used for movement and vision,
2. **Spot CORE** - A computer on Spot's back, used to send commands to the robot,
3. NVIDIA Jetson Xavier NX (**Jetson**) used for object recognition and image processing,
4. Kinova Gen3 Manipulator (**Kinova**), used for grasping and vision.

Figure 3.1 outlines at a high level how the four components are organised. The configuration is divided into the control system and hardware group. The control system, as the name suggests, is in control of all calculations and planning, and commands the hardware to perform actions, such as walking and grasping.

The hardware group contains Spot and Kinova, which both have vision modules which the control system uses for object detection. Within the control system, Spot CORE and Jetson are situated. All four components will be explained further in subsequent sections.

The inputs, upon being received from the hardware group, are processed in the control system, and converted into control commands that get sent back to the hardware group. There is bidirectional communication between Spot CORE and Jetson, which perform control and object detection actions, respectively.

3.1.1 System Communication: ROS

The control system is constructed on top of ROS (Robot Operating System). ROS is known as "middleware", which provides a layer of abstraction between the operating system and the programs running on it. In ROS, each program is situated in a "node", which is an independent unit. Nodes on the same network communicate through messages over "topics", which are channels which organise messages sent between nodes. This decentralised structure means that it does not matter what device each node is on, as long as it runs on the same network as all other nodes.

This distributed nature is the primary reason why ROS was preferred for this application. There is a continuous net of communication between the four devices (Spot, Spot CORE, Jetson and Kinova), which must exchange different types of data asynchronously for the pipeline to function. This system would have been difficult to implement without a system like ROS.

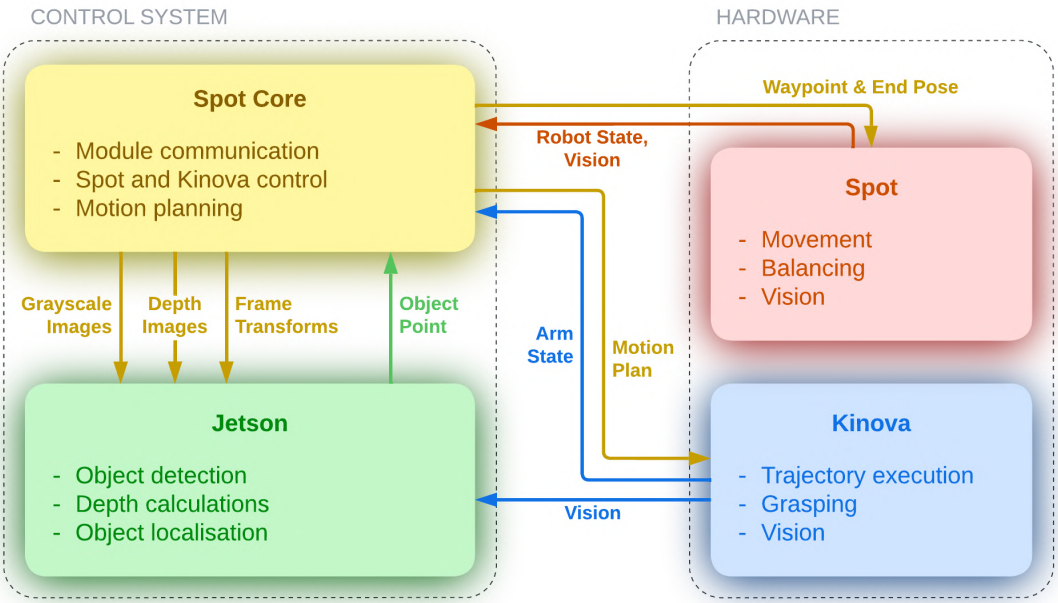


Figure 3.2: High-level overview interactions between system components

Some other alternatives exist, such as LCM from [Huang et al., 2010], but ROS was chosen due to the high volume of readily available learning resources, libraries, software support and ease of use. In this project, all nodes running on all three machines communicate with messages through dedicated topics.

3.1.2 Docker

All ROS nodes in Spot CORE run in a Docker container, which allows for efficient management of dependencies and isolation of systems. In addition to the dependency management, Docker enables scalability and reproducibility. Researchers working on similar topics will be able to use the same Docker images to build their containers, which will reduce the time spent setting up dependencies, speeding up future research.

Docker was not used in the Jetson, since almost all of the dependencies are proprietary NVIDIA packages already installed on the system when the NVIDIA JetPack software kit is flashed. This removes the need for Docker for project reproducibility. Additionally, flashing the same version of JetPack results in the same version of the dependencies being flashed, which removes the need for Docker for dependency management.

3.1.3 Component Interactions

With the knowledge of ROS and communication over topics, it is possible to outline in greater detail the functions of each component, as well as the interaction between various components. Figure 3.2 introduces the functions of, and interactions between, each of the four components.

This figure shows how the control system receives inputs from the hardware group and processes them to produce control outputs that are sent back to the hardware group. The functions and interactions outlined in this figure will be explained in detail in Section 3.2, Section 3.3 and Section 3.4, which concern each of the system components.

3.2 Hardware Group

3.2.1 Spot

BostonDynamics Spot is an advanced quadruped robot able to perform balancing, localisation and navigation tasks automatically and effectively. It does this with the help of the 8 cameras surrounding it, which provide 360-degree vision and depth input to the robot. All input gets processed by Spot's onboard computer, which uses the information to localise the robot in its surroundings, process commands coming from users and command the robot to navigate and balance itself. Spot's onboard computer is not open for development [BostonDynamics], as all of Spot's software is proprietary.

Spot's functions are as follows:

1. Provide depth and vision information to all other components using its eight visual and depth sensors,

2. Perform localisation and provide a localisation frame to all other components (`vision_odometry_frame`),

3. Provide frame transform and joint state information to all other components for object localisation,

4. Receive commands from Spot CORE and move to the desired poses.

3.2.2 Kinova

The Kinova Gen3 is a robotic arm with 7 degrees of freedom. It is controlled through its embedded controller, which can be interfaced with through the Kinova Kortex API. In addition to performing grasping tasks, it also has a vision module (Intel RealSense D435) on its gripper which captures RGB-D images for control and vision tasks.

Kinova's functions are as follows:

1. Provide depth and vision information to all other components using its vision module,
2. Provide transform and joint state information to all other components for object localisation,
3. Receive commands from Spot CORE and perform a grasp.

3.2.3 Spot and Kinova Vision Modules

The reader might have realised that both Spot and Kinova have overlapping roles in terms of vision. The system uses visual and depth inputs from both Spot and Kinova to detect and localise objects. This is due to the differences in the vision modules, which are compared in Table 4.1, in the Implementation chapter.

These differences will be explored in depth in Subsection 4.1.3. It will then become clear that the cameras serve different purposes, and that it is easier to recognise objects farther away using Kinova's vision module, whereas Spot's cameras are better when used to accurately localise nearby objects. This is due to Spot's cameras having a lower resolution but a wider field of view, and only capturing fisheye, grayscale images. Therefore, it is necessary to use both vision modules at different stages: Kinova's vision module when far from the object, and Spot's vision module after approaching the object.

3.3 Jetson

In order to perform an autonomous grasp, the system must be able to detect objects to pick up. To be used on this task, the NVIDIA Jetson Xavier NX was chosen. The Jetson is a portable computer, created for use in AI and ML tasks. Its size, low power consumption and high AI performance make it a good choice for this system.

The Jetson receives as input visual and depth images from Spot's and Kinova's vision modules to perform object detection. In doing so it creates bounding boxes for the objects, which it uses along with depth images to determine their positions in three-dimensional space. As an additional input, it also receives frame transformations from Spot. These are transformations that tell the Jetson how to transform object coordinates from the point of view of the cameras to a common frame of reference.

As a result, Jetson returns to Spot the transformed coordinates of the detected object in three-dimensional space. These provide information to Spot of the object's location relative to itself. This information will be referred to as *object localisation* for convenience, even though in computer vision this term refers to only the operation of drawing a bounding box around an object in a two-dimensional image.

3.3.1 Why Use an External Computer?

There are several constraints associated with performing object detection. The most pressing is that the object detection must be performed in real-time. This means that the detection must occur very quickly. Fulfilling this constraint requires one of two solutions:

1. Use a lightweight model which does not require large amounts of computing power, or
2. Perform the detection on a powerful computer with free resources dedicated for this task.

Options were considered that would run on Spot's CPU, such as [Chen et al., 2019], [Tan et al., 2019] and [Li et al., 2018]. These options are state-of-the-art models which focus on efficiency to increase performance when running on a CPU with low resources. However, these options all try to tackle the trade-off between detection accuracy and computation resource consumption [Li et al., 2018]. Since performing object detection both fast and accurately is crucial for the success of the project, using a larger model on powerful external GPU was preferred.

3.3.2 Object Detection With DetectNet

DetectNet is a neural network model architecture for object detection [Tao et al.]. It is also the name of a ROS node in the "ROS deep learning" package by NVIDIA, which uses various neural network models other than DetectNet for object detection. This node is convenient, and runs natively on the Jetson, using NVIDIA's JetPack software package to make use of the Jetson to speed up object detection. Additionally,

the pre-trained models provided by NVIDIA excel at general object detection - recognising a wide variety of objects (91 labels in the COCO dataset) accurately.

Working with grayscale images A challenge associated with object detection in this project is the inputs. Spot's cameras produce grayscale images, which reduce DetectNet's performance. This reduction is to such an extent that after several minutes of moving the teddy bear around in front of the grayscale camera, the model was not able to recognise the bear for more than a few frames. As a solution, the model was re-trained using transfer learning, on a dataset created from around 1300 pictures of the Personal Robotics Lab's teddy bear, which is the object used as a proof of concept for this project.

Transfer learning greatly improved DetectNet's performance on close-up pictures of the teddy bear from Spot's cameras. The self-trained model was able to recognise the teddy bear reliably within 1.5 metres of the camera. The self-trained model's performance is discussed in detail in Subsection 5.1.1. Differences in performance between the two models is highlighted in Figure 3.3.

(a) Pre-trained model (b) Transfer learning model

Figure 3.3: Comparison of object recognition results on grayscale and fisheye Spot camera, using pre-trained and custom-trained SSD-MobileNet-v2 models

How about detecting object poses? Based on the conclusion reached in Subsection 2.2.3, the project has decided to proceed with working only with a teddy bear for recognition. This object has distinct features which make it suitable for detection in grayscale images. Furthermore, it is soft and deformable, which removes the risk of dropping or breaking it. This relaxes the requirements on the gripper pose required to grasp the object, which removes the need to estimate the object pose.

As a result of the selection, the arm will be able to grasp the object regardless of the direction the end-effector approaches the object from, as long as the generated grasp pose is kinematically feasible.

3.4 Spot CORE

Since Spot's onboard computer is not open for development, an additional computer is required to develop code and run it on Spot. For this task, on Spot's back is a computer, called Spot CORE, which is used to run all developed software locally on the robot. In the project, this computer is responsible for all control of the systems, including processing the object coordinates and creating a motion plan for Kinova and and a waypoint for Spot.

Spot CORE performs the following functions in the system:

1. As ROS Master, enable communication between ROS nodes,
2. Interface with the BostonDynamics API to expose images (visual and depth) and Spot robot state to the ROS network,
3. Interface with the Kinova Kortex API to expose images (visual and depth) and arm robot state to the ROS network,
4. Given the object position, determine a grasp pose and motion plan for Kinova,
5. Given the object position, determine a final pose (waypoint) for Spot.

Spot CORE handles all communication with the hardware group, by providing access to sensor information (functions 2 and 3), and giving commands to the hardware group (functions 4 and 5). In this way, Spot CORE sits in multiple stages of the pipeline. The following sections will detail each of the functions listed above.

3.4.1 Interfacing with BostonDynamics API

BostonDynamics have created an extensive API to access and manipulate Spot's functions. Although this API does not provide control access to individual joints of Spot, it allows the developer to access all camera feeds and provide movement and navigation commands to Spot, among other functions.

Why Spot's cameras? As stated in Subsection 3.3.2, Spot's cameras are handicapped for the purposes of this project: The lenses are grayscale, fisheye and rotated. On top of this, there is no pointcloud provided by Spot's cameras, meaning

that the object position must be manually converted from camera coordinates to world coordinates. Furthermore, fixing image rotation is simple, but fisheye distortion could affect the accuracy of depth estimation and the lack of colour could make object detection more difficult.

In contrast, the Kinova arm not only has a higher-definition, undistorted, RGB camera, but it also has the added benefit that its Kortex API automatically exposes its images to the ROS network. In spite of these benefits, the project also makes use of Spot's two front cameras, in addition to the arm's vision system. This is only the case when Spot is within grasping range of the arm, and has three main reasons:

1. **Field of view:** As Spot is approaching an object, the arm moves around the robot, and the camera faces whichever direction the arm is facing. Thus, it is likely that an object in front of Spot will not be in the arm's field of view, or that the arm will lose sight of the object before the grasp. In contrast, Spot's cameras are fisheye and provide a much larger field of view (Will be specified in Section 4.1), leading to a larger probability that the object in front of Spot will be in the field of view of one of Spot's two front cameras.
2. **Scalability:** As Spot has 360-degree vision, its other cameras can be integrated to provide object detection capabilities that will be impossible with the arm's single camera. This can be implemented as a future objective to enable exploration.
3. **Aiding future research:** A stationary grasping pipeline has already been implemented with Kinova's vision module by Imperial's Personal Robotics Laboratory. However, Spot's cameras have not been integrated into this pipeline and doing so will aid and speed up future research in the topic.

3.4.2 Motion Planning for Kinova and Spot

With the object localised, Spot CORE must create a waypoint for Spot and a motion plan for Kinova. The motion plan must be constructed assuming Spot has already reached its final pose. Spot CORE can then send messages to both Spot and Kinova to start executing the motion plans at the same time.

How do we plan the arm's motion? The three-dimensional object position is first fed into a *grasp pose generator*, which generates a suitable pose (position and orientation) for the end-effector of the arm. Then, this final pose is plugged into a *motion planner*, which plans a trajectory from the arm's current state to the goal pose, taking into account specified constraints, such as the floor plane and surrounding obstacles.

Choice of grasp generator Based on the research conducted in Section 2.2, the author had multiple choices for grasp generation, ranging from MoveIt Grasps [MoveIt!,

b] and MoveIt Deep Grasps [MoveIt!, a] to PrendoSim [Abdlkarim et al., 2021], which is a more advanced grasp generator based on proxy hand robots.

Initially, the preferred option was MoveIt Deep Grasps, which is a complete pipeline on its own. It handles object detection, pose detection and grasp generation. This library would have run on the Jetson and entirely removed the need for DetectNet. This was, however, discarded due to compatibility issues, where the library could not run due to other dependencies on the Jetson.

Other grasp generators, such as MoveIt Grasps, were discarded due to the reasons given in Subsection 2.2.3. Instead, the `spherical_grasps_server` script from the `tiago_pick_demo` library [Pal-Robotics] was used. This is a grasp generator written in Python, and instead of a 6-dimensional object pose, it requires an object sphere, or the object's position and approximate size, as an input. This removes the need for calculating an object pose, which is convenient and reduces computational costs.

Choice of motion planner Based on the research conducted in Section 2.3, the author opted with the OMPL motion planner from the MoveIt library, due to its superior performance and efficiency.

3.5 Contributions

- A ROS package `auto-grasp` with the following nodes that support the implementation of a dynamic grasping pipeline:
  - A node to interface with the BostonDynamics API to rotate images from Spot's fisheye camera feeds before publishing them to ROS topics,
  - A node which determines an object's position in three-dimensional space given a bounding box, depth image and camera intrinsics,
  - A node which moves Spot to a distant detected object and oversees grasping actions,
  - A node which plans a trajectory for a detected object, taking Spot's movements into consideration.
- A Docker image and Makefile that allow for easy reproducibility of the project.

The last item in `auto_grasp` was developed as a modification of a planning node already implemented by Imperial's Personal Robotics Laboratory.

3.6 Analysis of the Design Process

The system design has changed considerably since the start of the project. One of the most significant problems faced by the author was compatibility. Compatibility issues between certain ROS modules and devices led to the selection of different modules, which sometimes even led to significant alterations in the system pipeline.

An example of this is the object detection framework used. The author had initially decided to pursue the DepthAI library with the OpenCV AI Kit, which works out-of-the-box for object detection. A comprehensive grasping pipeline was designed with this component in mind. However, the OpenCV AI Kit would have been connected to Spot CORE and would require access from within a Docker container. Due to incompatibility issues of `udev` rules and Docker, the image feed could not be accessed and this component was removed completely from the plans and the author switched to DetectNet.

Another reason for changes was a limitation in hardware. A fitting example is the use of Spot's cameras for all object detection tasks. This would have allowed for 360° vision in object detection, which would have enabled exploration tasks and made the project more comprehensive. However, due to the resolution and field of view of the cameras, the vision module was unable to recognise distinctive objects farther than about 2 metres away from the robot. Therefore, the pipeline was altered to use Kinova's vision module for object detection with the object not yet in grasping range of the robot.

The final pipeline created resolves compatibility issues, and manages to accomplish the task of object detection at a distance, as well as more accurate object detection when close to the object. Therefore, throughout the project, the design evolved in a direction which enabled the success of the project.

Chapter 4

The Implementation

Chapter 3 introduced at a high level the components which make up the system. This chapter will explain with technical detail the implementation of each component.

In this chapter, the entire pipeline will be explained, with regard to each ROS node and topic, how they interact with and affect each other, as well as technical details of each implementation.

Figure 4.1 brings into view the main ROS nodes in the system relevant to this project, their interactions with each other, how they interface with the BostonDynamics and Kinova Kortex APIs, and ultimately with the underlying hardware.

Upon inspecting Figure 4.1, the reader will notice that the control system is divided into three software ”units”: the action unit, interface unit and intelligence unit. These units are purely logical and aim to partition the system based on layers of abstraction. The lower the reader moves in the figure, the higher the software abstraction of the programs becomes. Below is a brief description of each unit.

- **Action unit:** The bridge between hardware and the control system. This unit allows the control system to access information from the hardware, as well as send commands to it.

- **Interface unit:** The bridge between the action unit and intelligence unit. Nodes in this unit extract the relevant information and functions from the APIs and publish them to ROS topics, allowing the intelligence unit to perform all necessary functions without ever having to manipulate the API directly.

- **Intelligence unit:** The control unit. All nodes which perform computations and generate outputs are in this unit. Outputs are sent from this unit to the interface unit.

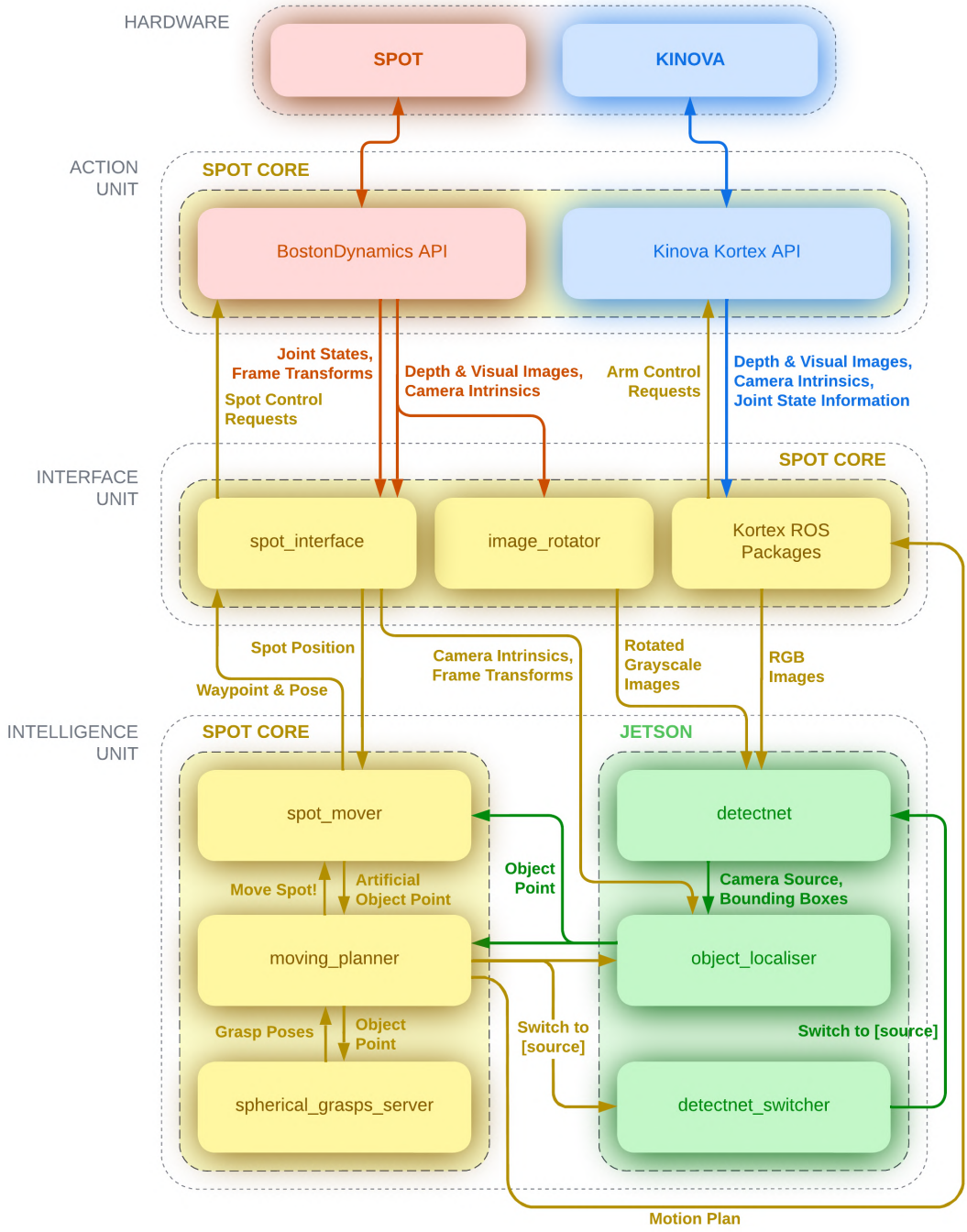


Figure 4.1: Detailed overview of system design, demonstrating ROS nodes and interactions between them

The flow of inputs is downwards in the figure, and the flow of outputs is upwards. Therefore, the data-flow follows a U-shaped pattern through the system.

The subsequent section will go into detail about the technical requirements and

system dependencies of each component, followed by sections explaining the data flow in the pipeline and the reasoning and software implementation of each node. The implementation will be explained on a per-node basis rather than per-unit, as this structure better follows the flow of data through the system.

4.1 System Dependencies and Technical Details

4.1.1 Spot CORE

All nodes on Spot CORE run in a Docker container, the image for which is derived from the `spot_prl` Docker image located in the `PRL-Dockerfiles` repository in the Imperial College London GitHub organisation. This Docker image was modified to contain all dependencies required by the project. All repositories from which code has been utilised will be listed in Chapter A.

Additionally, the BostonDynamics SDK arrives pre-installed on Spot CORE. Thus, all software required to control Spot exists on the device. The Docker image for Spot CORE uses the following packages to run all ROS nodes required:

- Ubuntu 20.04 Focal
- ROS Noetic
- Python 3.7
- BostonDynamics Spot Python Packages
- OpenCV
- MoveIt! Packages
- Kinova Kortex & Kortex Vision APIs

4.1.2 Jetson

The Jetson runs from an SD card flashed with NVIDIA's JetPack 4.6.2 software package, which includes, among other packages:

- Ubuntu 18.04 Bionic

- NVIDIA drivers
- TensorRT
- cuDNN
- CUDA

This software prepares it to run inference with neural network models. However, to allow it to run the DetectNet node and perform transfer learning on the author's Spot grayscale teddy bear dataset, more packages must be installed. The following packages are installed directly onto the device rather than in a Docker container, as the author has found that containerisation interferes with input devices by preventing `udev` rules from functioning properly.

- ROS Melodic
- Python 2.7
- PyTorch
- The `ros_deep_learning` package, which includes the DetectNet ROS wrapper

4.1.3 Hardware Group: Spot and Kinova

Out of the box, Spot and Kinova come pre-installed with BostonDynamics and Kinova drivers, respectively. Therefore, there is no need for any installations on the movement unit. However, the camera specifications of Spot and Kinova are stated in Table 4.1, as these specifications directly affect design choices.

Property	Spot	Kinova
Distortion	Fisheye	Undistorted
Visual Resolution	480 x 640	1280 x 720
Colour	Grayscale	RGB
Depth Range	4m	6m
Field of View	360°	72°
Pointcloud	No	Yes

Table 4.1: Camera specification comparison of Spot and Kinova's vision modules

It is evident from the table that the Intel RealSense camera on the Kinova arm has a narrower field of view, but is higher resolution. Furthermore, its depth range is

50% larger than Spot's cameras. These properties makes it more suitable for use when the robot is far away from the object. Furthermore, the images produced by this camera are RGB, as opposed to Spot's grayscale cameras. This enables the use of pre-trained DetectNet models.

Finally, the availability of a pointcloud makes it easier to retrieve object depth in the real world. However, this was not used, as obtaining this information from a depth image and transforming it to world coordinates with camera intrinsics is more efficient and faster: The pointcloud is calculated for every pixel in the image, but by using camera intrinsics and the depth image, the world coordinate calculations are performed only for the region of interest (9 pixels around the bounding box centre).

4.2 The Pipeline

This section aims to explain the dynamic grasping pipeline, with a subsection for each ROS node. The nodes will be described in the order of data flow in the system. The inputs (visual and depth images and robot states) get transformed by the control system into a waypoint for Spot and a motion plan for Kinova. The intermediate nodes and steps are described in the subsequent subsections.

4.2.1 ROS Node: `spot_interface`

The BostonDynamics API allows developers to get the information from the robot and send commands to the robot. The developer can get images, camera info, robot time, frame transforms and robot joint states. The developer can also send commands such as self-righting, trajectory, velocity, standing and sitting.

`spot_interface` is an interface between the BostonDynamics API and the ROS network, and it exposes the aforementioned commands and information to the ROS network. This section will use two examples to demonstrate how data flows **from the API to ROS** and **from ROS to the API**.

This node is not the author's contribution to this project - it is part of the `prl_spot` ROS package previously developed at Imperial's Personal Robotics Laboratory (PRL). However, the node is crucial to the functioning of the system, which is the reason behind its inclusion in the report. Furthermore, this node was modified by the author to provide additional functionality. The modifications will be explained in this section.

Data: BostonDynamics API → ROS

The node retrieves information from the API using its functions. For example, the source code in Listing 1 initialises the API's image client, which retrieves images

from Spot's cameras. This example is chosen for its simplicity and significance in the data flow of the control system.

```
import bosdyn.client
from bosdyn.client.image import ImageClient

class AsyncImageService(...):
    ...

class SpotInterface:
    ...
    def __init__(self, config):
        self.sdk = bosdyn.client.create_standard_sdk('spot_ros_interface_sdk')
        self.robot = self.sdk.create_robot(config.hostname)
        self.image_client =
        ↪  self.robot.ensure_client(ImageClient.default_service_name)
        ...
        # Use the image_client to retrieve images from Spot's image sources
        ↪  asynchronously.
        self._front_image_task = AsyncImageService(self.image_client,
        ↪  callback=self.FrontImageCB, ...)
        ...
        @property
        def front_images(self):
            return self._front_image_task.proto

        def FrontImageCB(self, results):
            data = self.front_images
            if data:
                image_msg0, _, _ = self.getImageMsg(data[0])
                self.frontleft_image_pub.publish(image_msg0)
            ...
```

Listing 1: Example of data extraction from the BostonDynamics API

Listing 1 shows the API being used to create the `sdk`, `robot` and `image_client` objects, which help retrieve images from Spot's cameras asynchronously. Whenever images are received, the `FrontImageCB()` callback is called. Here, the images are placed in ROS messages constructed in the (not shown) `self.getImageMsg()` function. Finally, the publisher publishes the image messages. Data retrieval works with this principle for all forms of data: The data (camera info, transforms, etc) is retrieved from the API and published in a similar manner.

Commands: ROS → BostonDynamics API

To demonstrate how the interface passes commands to the BostonDynamics API, an example `trajectory_cmd_srv()` will be used. This function is a ROS service handler, and is called when the ROS service `trajectory_cmd` is called. The service allows users to set a waypoint for Spot to move to.

`trajectory_cmd_srv()` is the function that was modified by the author. Before the modification, users could only send a goal position for Spot. The function was modified to also accept a goal orientation in the form of a heading in degrees or a quaternion. Another modification was made to slow down Spot as it is executing a trajectory command, to increase its stability and accuracy.

The function is divided into two snippets for better readability.

```
class SpotInterface:
    ...
    def trajectory_cmd_srv(self, trajectory):
        for pose in trajectory.waypoints.poses:
            x, y = pose.position.x, pose.position.y
            o = pose.orientation
            # A heading can be encoded in a quaternion message
            if o.x != 0 and o.y == 0 and o.z == 0 and o.w == 0:
                heading = o.x
            # Otherwise, treat as a quaternion
            else:
                heading = self.quat_to_euler(o).roll
            ...
```

Listing 2: Example of command passing to the BostonDynamics API, part 1: Orientation parsing

In the BostonDynamics API, a robot orientation must be supplied as a heading. Listing 2 demonstrates how the heading is extracted from the trajectory messages. The trajectory message has an orientation field in the form of a ROS quaternion message:

```
geometry_msgs/Quaternion
    std_msgs/Int x
    std_msgs/Int y
    std_msgs/Int z
    std_msgs/Int w
```

In a quaternion, it is unlikely for all fields except for `x` to be zero, as determined by experimentation: After checking over 200 instances of Spot's and objects' poses, this state was never encountered in a pose. Thus, the state is used to encode a heading in degrees in the `x` field of the quaternion. If this is the case, then `o.x` is treated as a heading in the function. Otherwise, the heading is extracted from the quaternion.

However, although experimentally deemed unlikely, this method is prone to errors if the encoding is somehow encountered in a quaternion. In the case that the encoding

occurs in a waypoint by mistake, it will only cause an error in Spot's final orientation rather than its position. Thus, this was deemed low-risk, as the robot is unlikely to unexpectedly strike bystanders by changing only its orientation.

This heading is then combined with the other parameters and supplied to the API to then be sent to the robot, as demonstrated in Listing 3.

```
from bosdyn.client.robot_command import RobotCommandBuilder, ...

class SpotInterface:
    def __init__(self, config):
        ...
        self.command_client =
        ↪  self.robot.ensure_client(RobotCommandClient.default_service_name)
        ...
    def trajectory_cmd_srv(self, trajectory):
        ...
            heading = ... # Calculate heading
            params = ... # Parameters with lower velocity
            cmd = RobotCommandBuilder.trajectory_command(goal_x=x, goal_y=y,
            ↪  goal_heading=heading, params=params, ...)
            self.command_client.robot_command(command=cmd, ...)
        ...
```

Listing 3: Example of command passing to the BostonDynamics API, part 2: API command request

4.2.2 ROS Node: `image_rotator`

Images produced by the two front cameras of Spot are rotated counterclockwise by 90 degrees ($\tau/4$ radians). The `image_rotator` node retrieves the images from the BostonDynamics API, rotates them clockwise by 90 degrees and publishes them to unique ROS topics. This is required to ease object detection on grayscale images with DetectNet. Although a new model was trained to detect images received from Spot's camera feeds, labelling upright images was a quicker and more efficient process.

In addition to retrieving images form the API like `spot_interface` the node also converts the raw images to OpenCV images to rotate and publish them.

4.2.3 ROS Node: `detectnet`

Part of NVIDIA's `ros_deep_learning` package, the DetectNet node uses a selection of models to perform object detection. The inputs and outputs are demonstrated in Table 4.2 and Table 4.3.

Input	Topic	Message Type
Grayscale or RGB Image	Input topic	`sensor_msgs/Image`

Table 4.2: DetectNet node input

Output	Topic	Message Type
Image with overlaid bounding boxes	`/detectnet/overlay`	`sensor_msgs/Image`
Detection results: bounding boxes, class IDs, confidences	`/detectnet/detections`	`vision_msgs/Detection2DArray`
Vision metadata: class labels parameter list name	`/detectnet/vision_info`	`vision_msgs/VisionInfo`

Table 4.3: DetectNet node outputs

In Table 4.3, one of the outputs is a list of detection results. Each result is of type `vision_msgs/Detection2D` which contains, among other information, the object class, bounding box centre coordinates (x, y) and bounding box size.

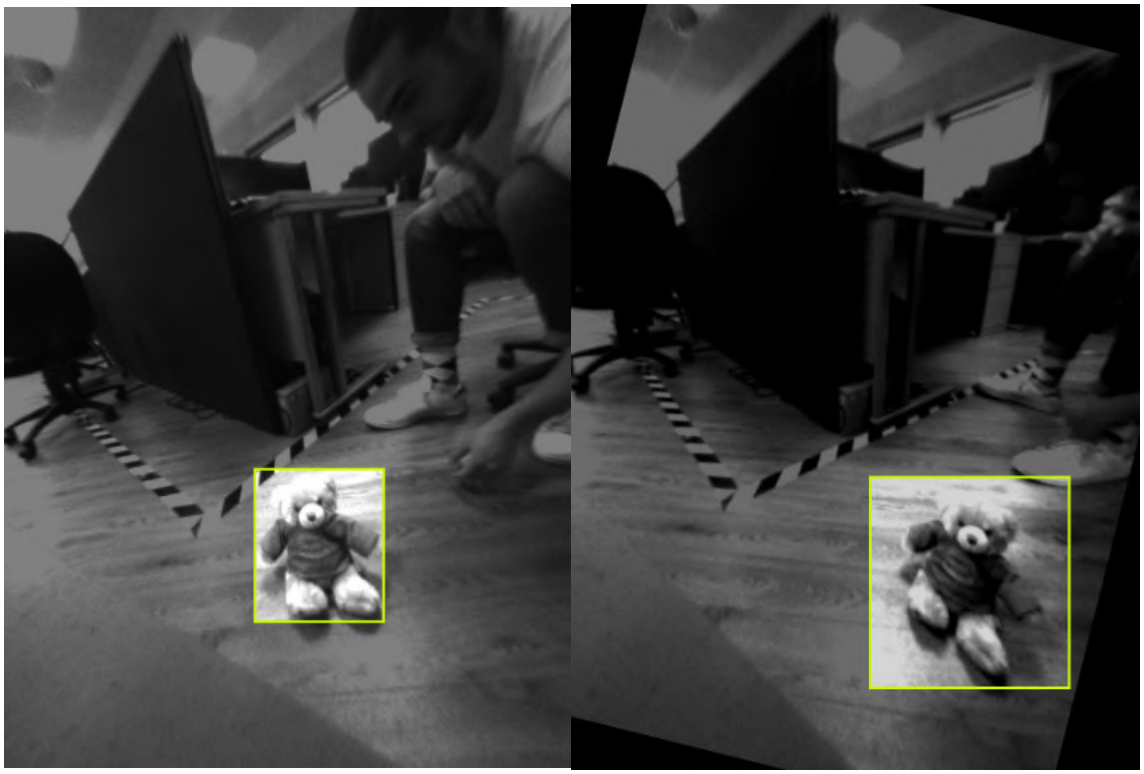
Because the two input sources used for object detection have very different properties (Subsection 4.1.3), two different models are required for object detection. Each model runs in its own DetectNet node. The models are as below:

1. The first is the SSD-MobileNet-v2 model included in the NVIDIA `jetson_inference` module, trained on the COCO dataset with 91 classes.
2. The second is a SSD-MobileNet-v2 model trained on the author's custom dataset of teddy bear images taken with Spot's grayscale, fisheye cameras.

The dataset that model 2 was trained on contains 1536 images of the teddy bear, annotated with bounding boxes in PASCAL VOC format. 512 pictures were taken by the author, and data augmentation with random 15-degree rotations and crops was used to extend the dataset to three times its original size. Figure 4.2 shows two examples from the dataset, one original and one with a random rotation.

By experimentation, the SSD-MobileNet-v2 model trained on the teddy bear dataset proved successful at recognising the teddy bear reliably up to a distance of about 1.5 metres from either front camera. However, the COCO-trained default model is able to recognise objects up to 3.5 metres away from the camera. This was also determined by experimentation, and by calculating distances when the arm is in its "home" configuration, where it is configured to stay in when recognising objects.

The experiments run to obtain these results are detailed in Subsection 5.1.1.



(a) Original sample (b) Rotated sample

Figure 4.2: Examples from the teddy bear dataset

The use of two separate models in the two subsequent stages necessitates either two `detectnet` nodes running at the same time, or quick switching of models and inputs. Running two nodes with two different models simultaneously was attempted, but due to both models requiring use of the GPU, this is not allowed by NVIDIA. Therefore, two launch files were created for nodes that run a node under the same name (`/detectnet`), but which look for models at different directories and different input topics.

Listing 4 shows the differences in the launch file arguments. Commented tags belong to the custom teddy bear model.

Since the node name is left unchanged, the first node automatically shuts down as soon as the second node is launched.

DetectNet source switching: `detectnet_switcher.py` ROS node

When Spot starts walking, the `moving_planner` node publishes a message to the `/switch_detectnet` topic. This performs two tasks, as shown in Figure 4.1.

First, it tells the `detectnet_switcher` node to automatically launch the second node. This node resides on Jetson, and uses the roslaunch API to launch the new DetectNet node under the same name.

```
<arg name="model_name" default="ssd-mobilenet-v2"/>
<arg name="model_path" default=""/>
<arg name="class_labels_path" default=""/>
<arg name="input_blob" default=""/>
<arg name="output_cvg" default=""/>
<arg name="output_bbox" default=""/>
<!-- <arg name="model_name" default=""/> -->
<!-- <arg name="model_path" default=teddy-bear-model-path/> -->
<!-- <arg name="class_labels_path" default=teddy-bear-labels-path/> -->
<!-- <arg name="input_blob" default="input_0"/> -->
<!-- <arg name="output_cvg" default="labels"/> -->
<!-- <arg name="output_bbox" default="boxes"/> -->

<node pkg="ros_deep_learning" type="detectnet" name="detectnet"
↪  output="screen">
        <remap from="/detectnet/image_in"
               to="/camera/color/image_raw"/>
        <!-- <remap from="/detectnet/image_in"
               to="/camera/frontleft_fisheye_image_rotated"/> -->
</node>
```

Listing 4: Differences in Kinova and Spot object detector launch files

Secondly, it tells object localiser node to switch image and camera info sources. The mechanisms behind this switch are explained in Subsection 4.2.4.

4.2.4 ROS Node: `object_localiser`

The `object_localiser` node combines bounding box coordinates with depth images, followed by using camera intrinsics to calculate real-world coordinates of the detected object from the camera's perspective. It then transforms this point to the vision odometry frame, which is the common motion planning frame, and publishes it.

The node's inputs are shown in Table 4.4. Here, either Spot's front right or front left camera is used. "`source`" is changed according to which input source is preferred.

Input	Topic	Message Type
Detection results: bounding boxes, class IDs, confidences	`/detectnet/detections`	`vision_msgs/Detection2DArray`
Spot camera depth image	`/depth/ + source + /image`	`sensor_msgs/Image`
Spot depth camera info	`/depth/ + source + /camera_info`	`sensor_msgs/CameraInfo`
Kinova camera depth image	`/camera/depth/image_raw`	`sensor_msgs/Image`
Kinova depth camera info	`/camera/depth/camera_info`	`sensor_msgs/CameraInfo`
DetectNet Node Image Source	`/switch_detectnet`	`std_msgs/String`

Table 4.4: Object localiser node inputs

The node processes all inputs in Table 4.4 to produce the output in Table 4.5.

Output	Topic	Message Type
Object Point: Vision Odometry Frame	`/object_point/odometry_frame`	`geometry_msgs/PointStamped`

Table 4.5: Object localiser node output

The object localiser node receives the bounding box from the same input topic (`/detectnet/detections`), but the bounding box must be interpreted differently based on the image source it is taken from. The treatment for both Kinova's and Spot's vision modules will be explained below.

Both image sources (Kinova and Spot) are modelled by the pinhole camera model. Thus, there exists a camera projection matrix P which represents the mapping of three-dimensional world coordinates to two-dimensional image coordinates according to the model. This matrix P is such that

$$P = \begin{bmatrix} k_x & 0 & p_x & 0 \\ 0 & k_y & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} k_x & 0 & p_x \\ 0 & k_y & p_y \\ 0 & 0 & 1 \end{bmatrix} [I|0] = K[I|0]$$

Where k_x and k_y are the x- and y-components of the camera's focal length, and p_x and p_y are the x- and y-components of its principal point.

When the depth images are received from either source, the appropriate camera's intrinsic properties are used to transform the image coordinates back to three-dimensional world coordinates from thee camera's frame of reference.

Listing 5 demonstrates this transformation, which is performed for images received from both sources. In the listing, the camera intrinsics are received from their respective topics presented in Table 4.4. Furthermore, the depth value is taken from the region of the depth image that the bounding box falls on.

Before the coordinates can be transformed, the bounding box coordinates must be transformed. The transformations are different depending upon the source.

Kinova vision module

The images received from Kinova's vision module are upright, removing the need for rotation to perform object detection. However, a complication is that the RGB and depth sensors on Kinova's vision module are separated by a distance of about four centimetres, which causes them to have different frames of reference. Furthermore, the depth camera has a resolution of 480×270, compared to the RGB sensor's 1280×720 resolution. These two differences necessitate rescaling of the bounding

```
class ObjectLocaliser:
    ...
    def camera_info_callback(self, camera_info_msg):
        ...
        self.depth_frame_id = camera_info_msg.header.frame_id
        self.width, self.height = camera_info_msg.width, camera_info_msg.height
        self.camera_intrinsics = {
            'focal_x': camera_info_msg.K[0],
            'focal_y': camera_info_msg.K[4],
            'principal_x': camera_info_msg.K[2],
            'principal_y': camera_info_msg.K[5]
        }
        ...

    def transform_image_to_world(self, depth, bbox_centre):
        tform_x = depth * (x - self.camera_intrinsics['principal_x']) /
        ↪  self.camera_intrinsics['focal_x']
        tform_y = depth * (y - self.camera_intrinsics['principal_y']) /
        ↪  self.camera_intrinsics['focal_y']
        tform_z = depth
        return tform_x, tform_y, tform_z
```

Listing 5: Camera to world coordinate transformation

box coordinates to match the colour resolution, and their transformation to the colour camera frame.

To transform the points, a ROS package `TF` is used, which constantly publishes and handles frame transformations between joints. The rescaling and transformation processes are shown in Listing 6. `self.listener` in this listing is a TF listener, which picks up published frame transforms.

TF publishes frame transforms asynchronously and `get_object_position_kinova()` is called from a callback, which makes it asynchronous too. For a transform to be available, line 15 in Listing 6 shows the script blocking until a transform is available. Furthermore, to ensure that transforms from the correct moment are used, Spot CORE's and Jetson's system clocks are synchronised using an NTP (Network Time Protocol) server and client set up on either machine. Details of setting up an NTP server are located in Section B.1.

Spot vision module

Unlike Kinova, Spot's vision module publishes depth images from the visual frame, which removes the need for rescaling and transformations. However, as described in Subsection 4.2.2, the images are rotated for object detection. Therefore, the bounding box coordinates x, y must be rotated to match the original image. This is done by the following simple transformation.

```
class ObjectLocaliser:
    def __init__(self, ...):
        ...
        self.listener = tf.TransformListener()
    ...
    def get_object_position_kinova(self, raw_depth_image):
        ...
        # Ratio to shrink bbox coordinates by
        colour_to_depth_size_ratio = self.width / 1280
        bbox_camera_frame.point = Point(
            x=self.raw_bbox_centre[0] * colour_to_depth_size_ratio,
            y=self.raw_bbox_centre[1] * colour_to_depth_size_ratio
        )
        try:
            self.listener.waitForTransform('kinova_camera_color_frame',
            ↪  self.depth_frame_id, ...)
            bbox_depth_frame = self.listener.transformPoint(
                    self.depth_frame_id, bbox_camera_frame)
        except ...
```

Listing 6: Pre-processing of bounding box coordinates from Kinova object detector

$$x, y = y, self.height - x$$

Post-processing

After the object point is received, this point is still with reference to the camera frame. The point is transformed to the `vision_odometry_frame` using the same method as in Listing 5, and then published to the output topic shown in Table 4.5.

This step is necessary due to the object position differences between frames, as shown in Figure 4.3.

DetectNet image source switching

The final feature of the object localiser node is the ability to automatically switch camera intrinsics and depth image sources when a message with the string ”spot” or ”kinova” is received in the `/switch_detectnet` topic. This change is performed in a ROS callback, as shown in Listing 7.

The change could also be performed using ROS Nodelets, which are designed to provide a way to run multiple algorithms on a single machine, in a single process. However, this method was not used, since switching the DetectNet and object localiser together was considered a simpler and clearer implementation.

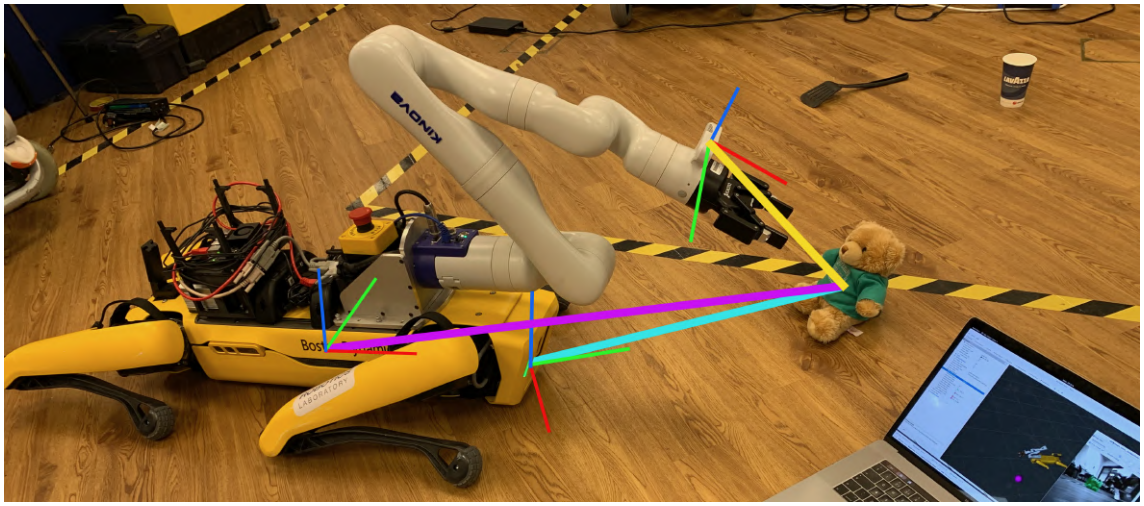


Figure 4.3: Differences in an object's position with respect to the Kinova camera frame (yellow), Spot front right camera frame (cyan) and vision odometry frame (magenta)

```
def switch_detectnet_callback(self, switch):
    # Reinitialise object with new source
    new_detectnet_source = switch.data
    assert new_detectnet_source == 'kinova' or new_detectnet_source == 'spot'
    if new_detectnet_source != self.current_detectnet_source:
        # __init__() handles setting the correct topics
        self.__init__(new_detectnet_source, self.depth_frame_id)
```

Listing 7: Object localiser: Handling of DetectNet source switching

4.2.5 ROS Node: `spot_mover`

The Spot mover node is responsible for coordinating the movements of the arm and Spot, as well as allowing the user to command the mobile manipulator to grasp a given object point. It handles the first stage of the grasp: Moving Spot towards the object and moving the arm close to the object. The node has inputs as shown in Table 4.6.

Input	Topic	Message Type
Spot kinematic state: Pose and joint states	`/kinematic_state`	`spot_driver/KinematicState`
Object point	`/object_point`	`geometry_msgs/PointStamped`
Move Spot signal	`/move_spot`	`std_msgs/String`

Table 4.6: Spot mover node inputs

Using the inputs, the node creates the outputs shown in Table 4.7.

Output	Topic	Message Type
Object proximity point	`/move_spot/proximity_point`	`geometry_msgs/PoseStamped`
Artificial object point	`/move_spot/artificial_object_point`	`geometry_msgs/PointStamped`

Table 4.7: Spot mover node outputs

In addition to the output messages, `spot_mover` also calls the `trajectory_cmd` service belonging to `spot_interface`. This allows it to move Spot to a goal pose.

As shown in Table 4.7, an "artificial object point" and a "proximity point" are created. The idea behind the grasp is that Spot is far away from the object, i.e. the object is out of the arm's reach. Therefore, any trajectory planned for the arm to reach the object point would be kinematically infeasible. This would cause the planning to fail.

The solution is to move Spot to the object's *proximity*, which is defined as Spot's centre (`base_link`) being 1.8 metres from the object which, considering Spot's length of 1.1 metres, leaves 1.25 metres between Spot and the object. The aim is to have the arm hovering approximately above the object when Spot stops moving. To achieve this, a trajectory must be planned for a point 1.25 metres in front of the robot and hovering 20 centimetres off the ground.

To create both these goals, `spot_mover` creates a vector directly from Spot to the object and calculates points on that ray 1.8 metres from Spot and 1.8 metres from the object, for the artificial object and proximity points, respectively.

These points are calculated as follows.

$$
\begin{aligned}
\boldsymbol{v} &= \boldsymbol{t} - \boldsymbol{s} \\
\boldsymbol{a} &= \boldsymbol{t} - \frac{\|\boldsymbol{v}\| - 1.8}{\|\boldsymbol{v}\|}\boldsymbol{v} + \begin{bmatrix} 0 \\ 0 \\ 0.15 \end{bmatrix} \\
\boldsymbol{p} &= \boldsymbol{s} + \frac{\|\boldsymbol{v}\| - 1.8}{\|\boldsymbol{v}\|}\boldsymbol{v}
\end{aligned}
$$

Here, $\boldsymbol{s} \in \mathbb{R}^3$ and $\boldsymbol{t} \in \mathbb{R}^3$ are Spot's and the teddy bear's positions, and $\boldsymbol{a} \in \mathbb{R}^3$ and $\boldsymbol{p} \in \mathbb{R}^3$ are the artificial object and proximity points, respectively. The artificial object point is raised (along the z-axis) by 15 centimetres, to allow the arm to hover over the object after trajectory execution. The created vector and points are shown in Figure 4.4.

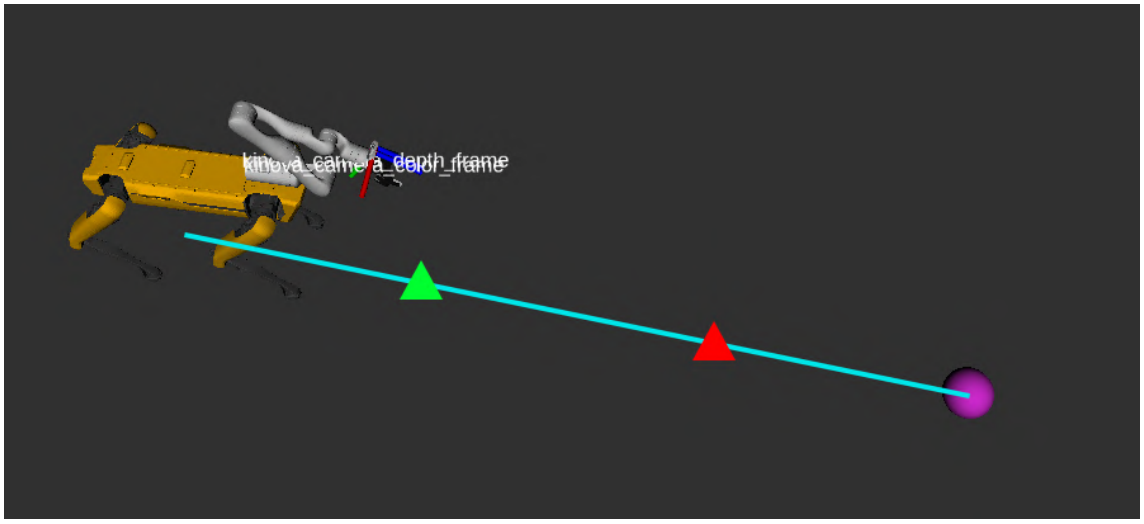


Figure 4.4: Constructed vector from Spot to the object (cyan), proximity point (red) and artificial object point (green), as generated by `spot_mover`

Along with the goal position, Spot also requires a goal orientation. For the second stage of the grasp, Spot's front left camera is used. This camera points slightly towards the right of the robot, which requires the robot to face slightly left of the object to have the object in sight of the camera. Therefore, the chosen orientation can be expressed as follows.

$$\boldsymbol{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, h = \arctan \frac{y}{x} + 0.2$$

This orientation h is a heading in radians, and is encoded as shown in Subsection 4.2.1 before being sent to the `trajectory_cmd` service. 0.2 rad is added at the end to make the robot face left of the object.

Moving Spot and pose history

When the proximity and artificial object points are calculated, the user is presented with three options, as shown in Figure 4.5. Listing 8 shows the actions taken upon the user choosing an option.

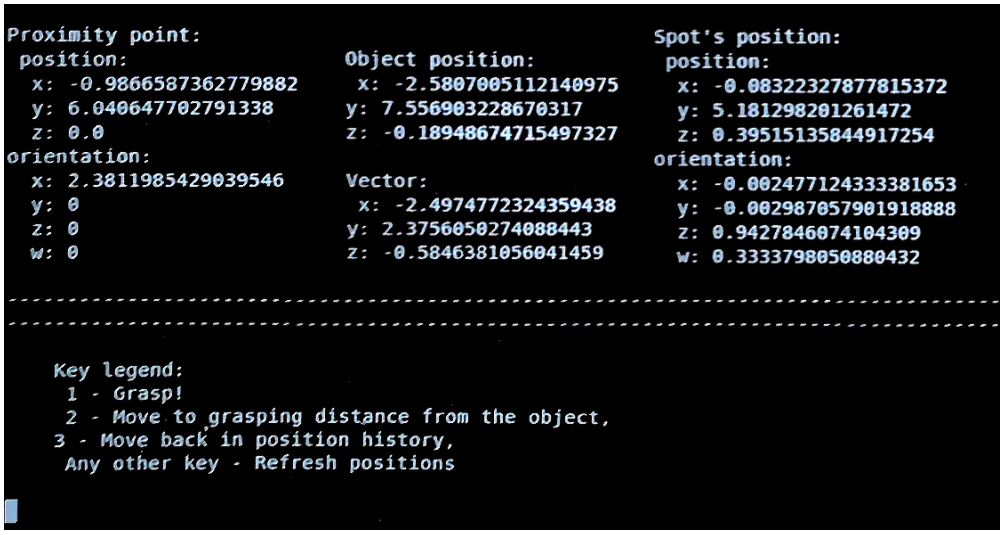


Figure 4.5: Spot mover user interface after object point is identified (Reshaped)

```python
class SpotMover:
    ...
    def object_point_callback(self, object_point):
        if point_to_move == '1' or point_to_move == '2':
            self.goal_pose_2d = self.proximity.pose
            if point_to_move == '1':
                self.artificial_object_point_pub.publish(
                        artificial_object_point)
        elif point_to_move == '3' and len(self.past_poses) > 0:
            self.next_poses.append(self.pose)
            self.goal_pose_2d = self.past_poses.pop().pose
        elif point_to_move == '4' and len(self.next_poses) > 0:
            self.goal_pose_2d = self.next_poses.pop().pose
        self.past_poses.append(self.pose)
        ...
```

Listing 8: Implementation of grasping and pose history in Spot mover node

Pose history is implemented by popping past poses from a stack. When grasping, the artificial object point is published to its topic. This signals `moving_planner` to start planning. When the arm is ready to start moving, `moving_planner` sends a message back to `spot_mover` on `/move_spot`, which initiates the `trajectory_cmd` service call, which moves spot to the proximity point, as demonstrated in Listing 9.

```
class SpotMover:
    def spot_move_callback(self, _):
        if not self.proximity:
            return
        self.trajectory.waypoints.poses = [self.goal_pose_2d]
        try:
            rospy.wait_for_service('trajectory_cmd', timeout=2.0)
            self.trajectory_cmd(self.trajectory)
        except rospy.ServiceException as e:
            print("Service call failed: \%s"\%e, end='')
            return
```

Listing 9: Sending trajectory commands to the trajectory command service from the Spot interface

4.2.6 ROS Node: `moving_planner`

As the final node in the system, `moving_planner` is responsible for generating grasp poses and performing motion planning. Additionally, it coordinates DetectNet camera source switching and performs the second stage of the grasp using the new object point received from Spot's vision module. The node's inputs and outputs are shown in Table 4.8 and Table 4.9.

Input	Topic	Message Type
Object point	`/object_point/odometry_frame`	`geometry_msgs/PointStamped`
Artificial object point	`/move_spot/artificial_object_point`	`geometry_msgs/PointStamped`

Table 4.8: Moving planner node inputs

Output	Topic	Message Type
Move Spot signal	`/move_spot`	`std_msgs/String`
DetectNet Node Image Source	`/switch_detectnet`	`std_msgs/String`

Table 4.9: Moving planner node outputs

`moving_planner` was constructed on top of the `planner` node created by PRL. The functions responsible for coordinating the grasp generation and motion planning were used with modifications.

Grasp pose generation with `planner`

The `planner` node uses the object position to generate an array of possible grasp poses using the spherical grasp generator discussed in Section 2.2 and Subsection 3.4.2.

Motion planning with `planner`

```
class Planner():
    ...
    def createPickupGoal(self, group="arm", target="TeddyBear",
            possible_grasps=[], links_to_allow_contact=None):
        pug = PickupGoal()
        pug.planner_id = "OMPL"
        pug.target_name = target
        pug.group_name = group
        pug.possible_grasps.extend(possible_grasps)
        pug.planning_options...  # Set planning options
        pug.attached_object_touch_links = ['<octomap>']
        pug.attached_object_touch_links.extend(links_to_allow_contact)
        return pug
```

Listing 10: Creating pickup goal for motion planning

Rather than using MoveIt's Python API, `planner` communicates with the "MoveIt pickup server". This is a ROS `actionlib` ActionServer. The `actionlib` library provides a standardized interface for interfacing with preemptable tasks. This is similar to a ROS service, but allows the user code to set a goal, track the process of the action through continuous feedback and receive a result.

To perform motion planning, the `/pickup` action server receives a PickupGoal, containing fields such as an array of grasp poses and the motion planner to use. The motion planner is set to OMPL, as discussed in Subsection 2.3.3. This goal is then sent to the pickup server. The construction of this goal is demonstrated in Listing 10. The motion planner choice and some planning options were modified from PRL's original `planner` script.

To perform motion planning, the planner requires knowledge of the planning scene. The required information consists of the position of the floor plane, joints of the arm, and the object to be grasped, as a collision object. The project assumptions, as stated in Section 1.3, state that there are no obstacles in the scene. Therefore, no other collision objects are required in the planning scene.

The implementations of the planning scene construction and pickup server interactions are shown in Listing 11.

Listing 11 was simplified to illustrate the pickup server interactions. The reader is drawn to line 16, where the script interfaces with the Kinova Kortex API to open the arm's gripper.

```
1  class Planner():
2      def __init__(self):
3          self.scene = moveit_commander.PlanningSceneInterface()
4          self.pickup_ac = SimpleActionClient('/pickup', PickupAction)
5          self.pickup_ac.wait_for_server()
6          self.sg = SphericalGrasps()
7          ...
8
9      def grasp_object(self, pose, object_class="TeddyBear"):
10         self.switch_detectnet_pub.publish(String("spot"))
11         ...
12         self.add_collision_objects(pose, object_class)
13         self.pick(pose, object_class)
14
15     def pick(self, pose, object_class):
16         self.kinova_gen3.reach_gripper_position(0.1)  # Open gripper
17         grasps = self.sg.create_grasps_from_object_pose(pose)
18         goal = self.createPickupGoal("arm", object_class, pose, grasps,
           ↪  self.touch_links)
19         self.pickup_ac.send_goal(goal)  # Send goal to ActionServer
20         self.spot_move_pub.publish(String("Time to move Spot!"))
21         ...
22         self.pickup_ac.wait_for_result()
23         result = self.pickup_ac.get_result()
24         ...
25         if self.first_stage_complete:
26             self.go_to_joint_position(known_joint_positions['home2'], ...)
27         self.kinova_gen3.reach_gripper_position(0.1)
28         self.first_stage_complete = not self.first_stage_complete
29         ...
```

Listing 11: Motion planning implementation with MoveIt Pickup server

`moving_planner`: The two grasping stages

As opposed to `planner`, `moving_planner` also adds the capability to coordinate Spot's movements and perform two grasping stages. The first grasping stage occurs when the artificial object point is received. In the artificial object point callback, `grasp_object()` is called, which calls `pick()`. After the trajectory is planned, line 20 in Listing 11 shows a string being published to the Spot move topic, which initiates Spot's movement at the correct time.

Furthermore, the reader is drawn to line 25, where a flag `self.first_stage_complete` is checked. This flag is only set if the first grasping stage has already been completed (If the current grasp is stage 2), and instructs the arm to return to its home position after the grasp. Finally on line 28, the flag is toggled, since the grasp has now been performed.

The second stage of the grasp is performed on the object point, since for this

stage Spot is at its final position. Therefore, the object point callback checks the `self.first_stage_complete` flag, and performs the second stage of the grasp if the flag is set.

4.2.7 Pipeline Setup and Grasp Flowchart

There are several nodes in the system, which need to be running for a grasp to be performed. The below list outlines the sequence of nodes that need to be launched to prepare the system for a dynamic grasp.

1. Spot driver package (`spot_interface`) (Spot CORE)
2. ROS Kortex drivers (Spot CORE)
3. Image rotator (Spot CORE)
4. Spot description (Spot CORE)
5. DetectNet node (Jetson)
6. Object localiser (Jetson)
7. Moving planner (Spot CORE)
8. Spot Mover (Spot CORE)

The 4th item on the list, Spot description, contains the Spot and Kinova URDF model descriptions which allow for motion planning. This package was previously developed by PRL and is used without any modifications.

After all of the above nodes are launched, Spot is placed at a distance of around 4 metres from the object, with the arm in the home position and facing the object. After this point, the grasp occurs as described in the flowchart in Figure 4.6.

In the flowchart, it is evident that the pipeline contains some errors that are possible to recover from. For example, if an object point cannot be calculated (Possibly due to lack of depth data at the position of the bounding box), or if trajectory execution fails due to joint limits, another object point can be calculated and the pipeline can continue. However, if an error is caused by the system itself, such as Spot not moving or Spot's or Kinova's camera feed not being available, the only method for recovery is to restart the system. If such errors do not occur, the pipeline results in a successful grasp.

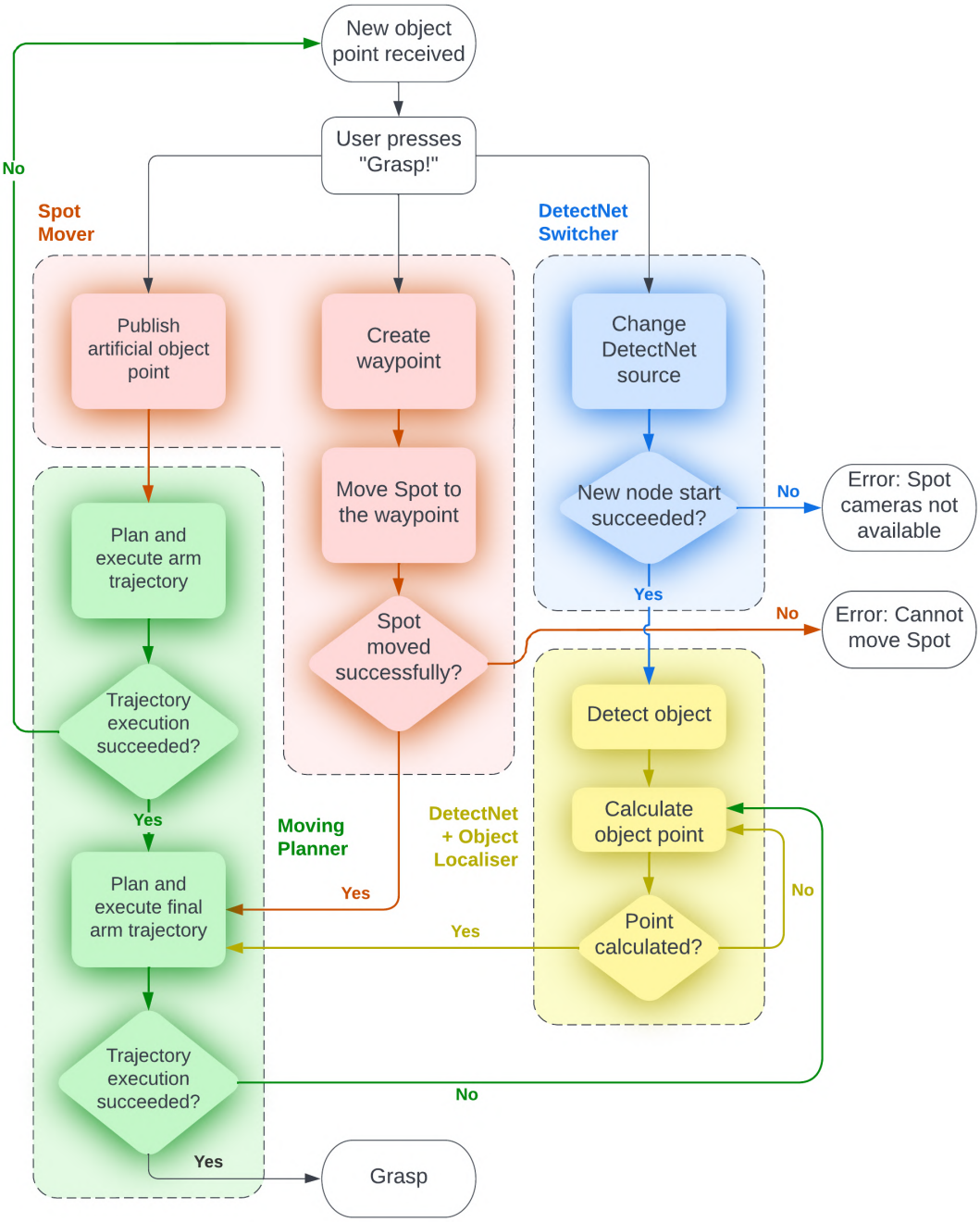


Figure 4.6: Flowchart showing the dynamic grasp process and involved nodes.

4.3 System Evolution

The design process of the dynamic grasp pipeline has evolved continuously since the idea's conception. In addition to the design changes explained in Chapter 3, the process itself evolved. This section aims to explain how the design, and the design

process, evolved throughout the project.

The initial development plan discussed in the interim report was well structured. First, only the object detection and localisation nodes would be developed on their own. Then, the object coordinated would be used to move a simulated arm to perform a grasp. After the simulation succeeded, the learnings would be transferred to the real arm. Finally, the arm would be placed on Spot for the development of the dynamic grasping pipeline. However, implementation requirements caused the design process to change entirely.

4.3.1 Stage 1: Jetson and RealSense

The first stage started as expected. ROS Master was run on a laptop computer rather than Spot CORE, the Jetson was connected to ROS Master, and an Intel RealSense camera was connected to the Jetson to handle vision and depth inputs. The DetectNet nodes were set up and tested. However, when the object localiser node was being constructed, the frame transforms became an issue: The images were being taken from an arbitrary perspective, since the camera could not be held stable.

If the calculated object point had been used to move a simulated arm, the camera's frame would not have been connected to any of the simulated arm joint states. Thus, the simulated arm stage was not possible, and was skipped.

4.3.2 Stage 2: Jetson and Kinova

The switch to the real Kinova arm allowed the use of the arm's vision module rather than a separate RealSense camera, and enabled the proper implementation of the object localiser. However, everything was still visualised in RViz and the arm was not moved until there was no risk that it would move to an unexpected position. When moving the arm, the planning was done with respect to the arm's `base_link`.

At this stage, a major design change in object localiser was observed. A major problem was that depth image and DetectNet detections were not synchronised. There was no information in the DetectNet output that specified which depth image is associated with the source image of the detections received. Initially, a script was written to modify thee DetectNet outputs to include the timestamp of their source image, synchronise the callbacks for the depth image and modified DetectNet detections, and process the correct depth image. However, this was expensive and complicated. Upon realising that in a few milliseconds the depth will not change significantly enough to cause a problem, the implementation was removed and the author opted to always use the latest depth images, asynchronously.

4.3.3 Stage 3: Jetson, Kinova and Spot

At this stage, the arm was placed on Spot. The planning frame was switched from Kinova's `base_link` to the `vision_odometry_frame`. This stage saw the development of spot mover and moving planner. Much like stage 2, all actions were first tested and visualised in RViz before being attempted on the real robot. Even then, some unexpected movements occurred, and the robot was stopped immediately.

Chapter 5

Testing and Results

Throughout the development of the dynamic grasping pipeline, performance bottlenecks were encountered. Some were explained in throughout the report, such as the inability of Spot's fisheye cameras to recognise objects far away, and Kinova's vision system having a limited field of view.

Such limitations necessitated design changes in the system. Irrespective of the design changes however, there are still factors which affect the system's performance. This chapter concerns the overall system performance, how each component affects the the overall performance, and the reason behind the performance figures.

The chapter will first go into detail about the performance of individual system components, followed by providing an overview of the components' effect on the overall system performance.

5.1 Testing System Components

This section explains the various components of the system that were tested, what was tested with each component, the methods used to perform the tests, how performance was evaluated and the test results. Test results outline under what conditions each component works, why each component works only under the given conditions, and the effect of the component's limits on the system as a whole. In the subsequent section, the test results are evaluated considering the system in its entirety.

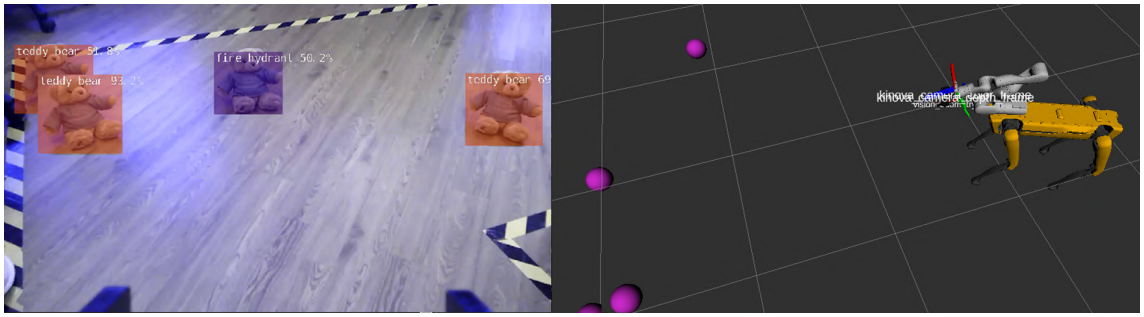


Figure 5.1: Teddy bear at the edge of the pre-trained SSD-MobileNet-v2's detection limits captured on Kinova's vision module and visualised in RViz, locations 1 through 4 from left to right

5.1.1 Object Detector

What was tested?

The object detector is responsible for finding and locating the objects that require grasping. Therefore, the reliability and range of predictions is crucial. Additionally, the location of the bounding boxes is paramount to locating the object, which in turn enables a successful grasp.

Testing maximum detection range

To test the maximum detection range, the object was placed in the detection frame, and moved further from the robot until the object detection stopped working reliably. At this point, distances with respect to the visual frame and base link (centre of Spot) were measured. This was repeated at four locations along the length of the image.

Kinova and pre-trained SSD-MobileNet-v2

Figure 5.1 shows the teddy bear various points of the edge of the Kinova object detector's reliable detection limit. In the figure, four teddy bear detections are overlaid on the same image. It is worth noting that the teddy bear is sometimes classified by DetectNet as a fire hydrant, but this behaviour is accepted as it is consistent.

It is evident from the three-dimensional simulation in the figure that the limit approximates a spherical perimeter. Table 5.1 presents L_2 norms of the recorded object detector limits, with respect to the the camera frame and base link - or the distance of each limit object from the camera and from Spot's centre.

The above results show that on average, the object detector is able to recognise objects up to 2.8 metres away from Kinova's camera and 3.4 metres from Spot's

Location	L_2 norm to camera frame (m)	L_2 norm to base_link (m)
1	2.900	3.507
2	2.651	3.256
3	2.885	3.486
4	2.801	3.321
Average	2.809	3.393

Table 5.1: L_2 norms of Kinova object detector detection limits along the image (1 to 4 from left to right)

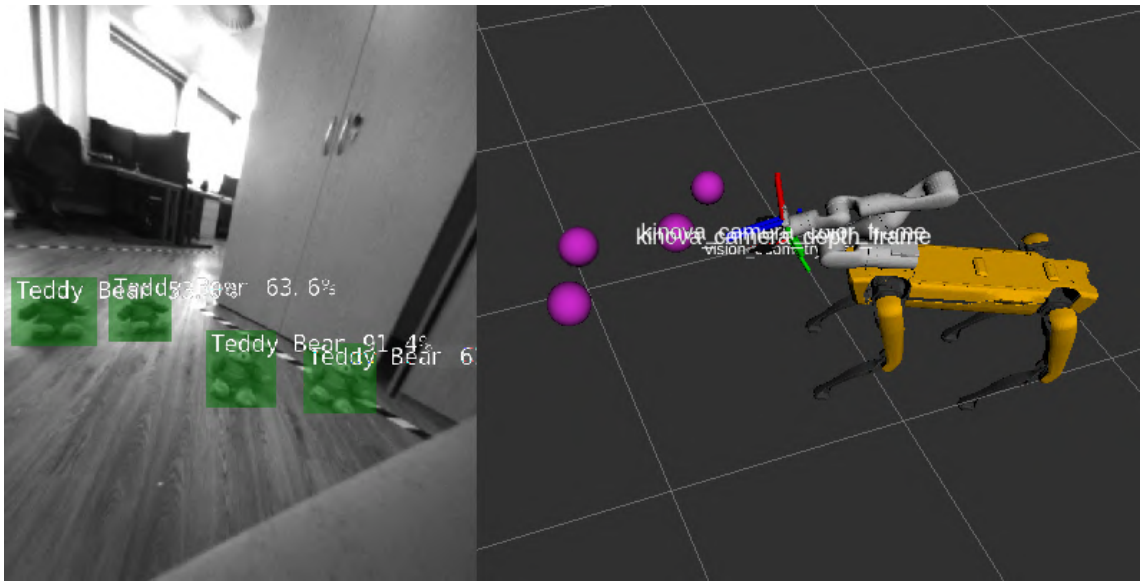


Figure 5.2: Teddy bear at the edge of the custom SSD-MobileNet-v2's detection limits captured on Spot's vision module and visualised in RViz, locations 1 through 4 from left to right

centre. Given Spot's measurements, the distance equates to around 3 metres from Spot's front end.

The above measurements were calculated with the arm at its home position, as shown in the simulation in Figure 5.1. The arm is able to reach 902 millimetres when fully extended, which would enable object detection up to 3.6 metres away from the robot.

Spot and custom teddy bear SSD-MobileNet-v2

Figure 5.2 shows the teddy bear placed at the edge of the Spot object detector's reliable detection limit. Due to the camera being a fisheye lens, the limit perimeter has a smaller radius than that of Kinova.

Table 5.2 presents L_2 norms of the recorded object detector limits, with respect to the the camera frame and base link - or the distance of each limit object from the

camera and from Spot's centre.

Location	L_2 norm to camera frame (m)	L_2 norm to base_link (m)
1	NaN	1.889
2	1.185	1.552
3	1.328	1.617
4	2.061	2.260
Average	1.525	1.830

Table 5.2: L_2 norms of Spot object detector detection limits along the image (1 to 4 from left to right)

There are a couple properties to note from the data.

First, the L_2 norm to camera frame at location 1 is NaN (not a number). This is because the grayscale image is 640×480, whereas the depth image is 424×240 which, even when scaled up to match the grayscale image, is a narrower aspect ratio. Therefore, detected objects close either side of the grayscale image do not have depth values associated with them.

When the object's depth value (distance to the camera frame) is not a number, how can its transform to base_link be a number? This is because the transform uses the last available point, which is from before the object was moved to its correct position, when it was within the depth image.

Second, the object detection limit is 1.5 metres from the camera frame and 1.8 metres from Spot's centre. This is much lower than the Kinova detector and is expected, due to the camera's lower resolution, fisheye distortion and lack of colour.

Evaluation of object detector ranges

As expected, the Kinova object detector has a superior range to that of the Spot object detector. However, an unexpected result was the narrower depth image aspect ratio, which causes objects detected at the image corners to not have depth values. This causes issues with object localisation for grasping, and requires the object to be inside a smaller window after Spot has moved to grasping distance of th object. This imposes tighter constraints around the required end pose of Spot after it has stopped moving. The tolerances of Spot's movements and the precision of the end pose are discussed in the subsequent sections.

Could we improve the Spot object detector range?

As shown in Table 5.2, the custom object detector model performs poorly over distances of 1.5 metres from the camera. Increasing this distance to about three metres would allow for the use of Spot's cameras throughout the pipeline, increasing the viewing range greatly.

In an attempt to fix the issue, a different object was tested: A more distinct, white "angry bird" plushie toy was used to create an object detection dataset of 2000 images taken with Spot's cameras. The dataset contains pictures of the plushie at a distance of 2 to 4 metres from the robot. Then, the dataset was used to train another SSD-MobileNet-v2 model.

Figure 5.3: Poor object detection results by custom angry bird model trained for use on distant objects using Spot's cameras

The model was able to recognise the plushie at a longer distance than the teddy bear model (shown in Figure 5.3), but the recognition was uncertain (about 50% confidence) and very unreliable. Therefore, the idea was not integrated into the system.

Testing object detector accuracy/precision

Object detector accuracy has a large impact on the success of dynamic grasps. During the first stage of the grasp, the object is up to 3.5 metres away from Spot. At this distance, a small mistake in the bounding box position could lead to large errors in object localisation and could cause the arm to end up at the wrong location. At the second stage, the shorter distance reduces the risk of making an error, but since this is the stage where the grasp must occur, even a small mistake could cause the arm to miss the object.

To test precision, an object was left at the same spot over a certain duration, the object's position was continuously calculated using the object localiser, and changes in calculated location were recorded across 1000 measurements, using a script that continuously records the uncertainties in the object point. The uncertainty was measured per axis (x, y, z), then averaged.

To test object detector accuracy, an object was placed at a known location in the

room, and its distance to the camera was measured with a tape measure. Then, the detector and object localiser were used to calculate the object's position relative to the camera frame. The detector results were then compared to the baseline.

This experiment was repeated for two locations each, using the Kinova and Spot object detectors. The results are presented in Table 5.3.

Location	Object detector	Average calculated distance (m)	Actual distance (m)	Relative error	Relative uncertainty
1	Kinova	2.36	1.64	43.9%	10.2%
2	Kinova	1.18	1.19	0.84%	12.0%
3	Spot	0.93	1.01	7.92%	9.61%
4	Spot	0.89	0.92	3.39%	7.27%

Table 5.3: Results of object detector accuracy and precision experiments

Object detector accuracy and precision evaluation

The results are generally consistent - the errors are under 10% and the uncertainties are around 10%. This is a high uncertainty, and over large distances, it will cause significant problems. For example, for an object 3.5 metres away, it might cause the robot to move 35 cm towards either side of the object, rendering the grasp unsuccessful.

Moreover, the relative error for the object at location 1 is 43.9%, which is very high. This has occurred because occasionally, the object point jumps to a wrong location. This is a bug and is thought to be caused by incorrect camera intrinsics switching, but no solution was found regardless of several attempts to fix it.

To prevent the bug from affecting the grasp success, the users are advised to visualise the object point in RViz and verify that the object point is at the correct location.

To improve the object detector accuracy and precision, cameras with more accurate depth measurements could be used, or multiple cameras could be used to measure the depth from multiple perspectives, locating the object accurately.

5.1.2 Stationary Grasping Pipeline

After the object has been recognised, two grasps must be performed. During the development of the system, some grasps were successful, while others failed. Therefore, it is important to determine the conditions under which a grasp is successful.

There are two main reasons why a grasp might fail. Either the object is out of the arm's reach, or there are no kinematically feasible trajectories from the arm's

current pose to the end pose required to grasp the object. A trajectory may be kinematically infeasible if there is a collision in the way, i.e. the joints do not allow the motion required or there is an object blocking the way.

In both cases, the grasp failure stems from the object placement. This section tests the stationary grasping pipeline to determine an array of object placements which result in a successful grasp.

What object placements will end in a successful grasp?

The grasping pipeline was tested with the object at various locations using Spot's camera. This camera was chosen due to the larger field of view available for testing. Furthermore, the object point (or artificial object point) is always within the area detectable by the Spot object detector, removing the need to perform tests with the Kinova object detector.

How were the tests conducted?

15 grasps were performed with the object in various points, most of which were edge cases. 7 of the 15 grasps failed, with the remaining 8 grasps defining the allowed grasping area. Snapshots from the 8 successful grasps are presented in Figure 5.4.

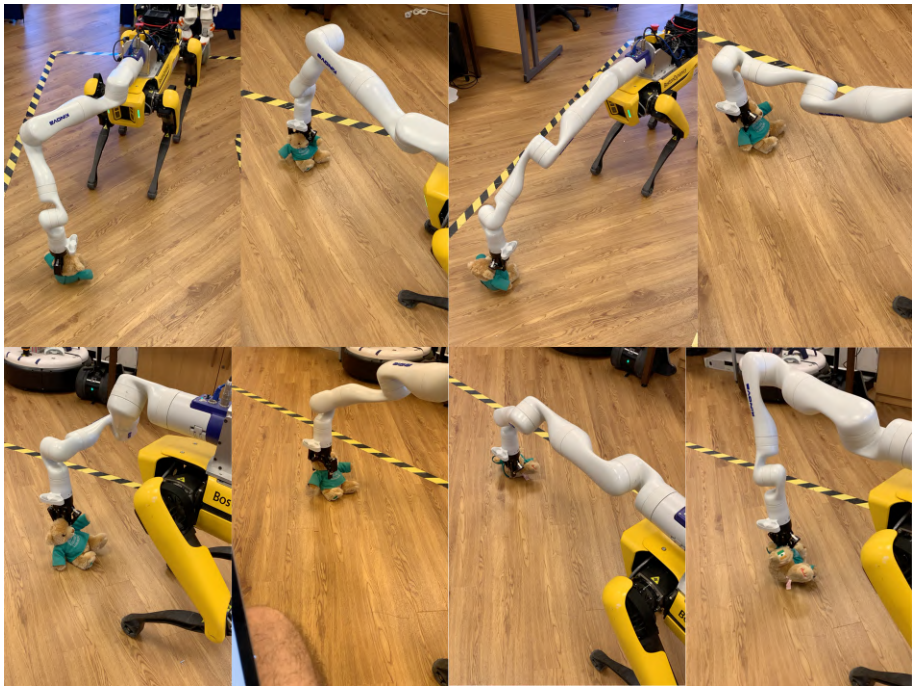


Figure 5.4: Examples of successful stationary grasps, where the object was placed inside the allowable grasp area

Consistent to the results from the object detector tests, the lack of depth data on either side of the image resulted in failed planning attempts when the object is too close to the image edges. Furthermore, the object must be within approximately 1

metre of the robot for a successful grasp. The above conditions lead to the grasp area highlighted in Figure 5.5.

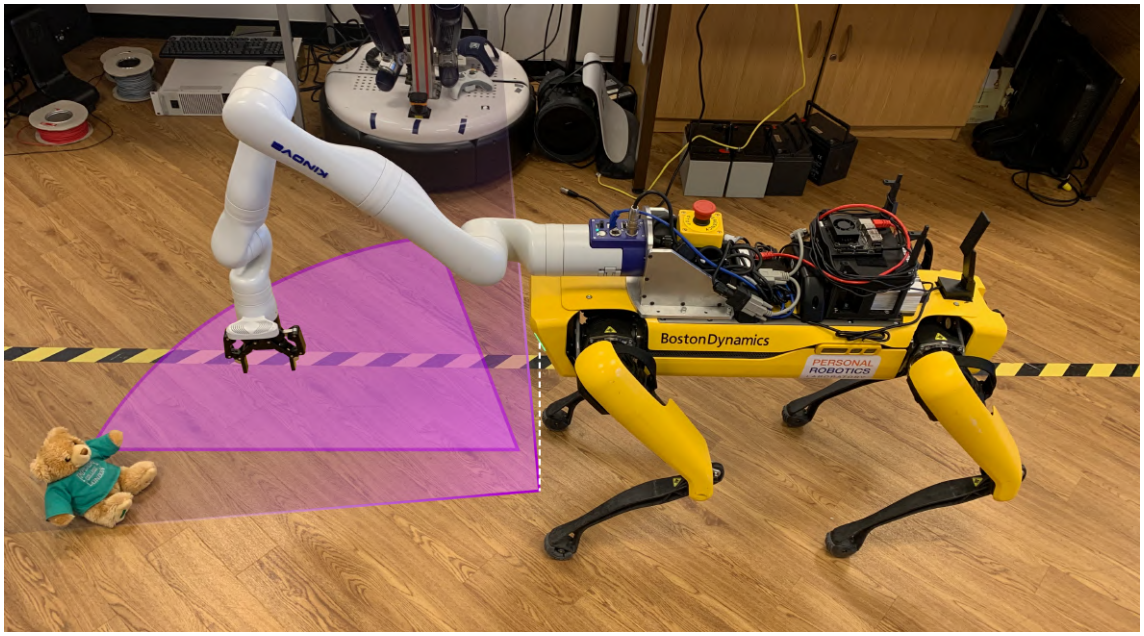


Figure 5.5: Allowable object placements for successful grasping. Light purple gradient shaded area represents viewing range of the front left camera, and the purple pie represents the area of allowable object placements.

In Figure 5.5, the large area shaded light purple represents a projection of the front left camera's viewing range onto the floor plane. Placing the teddy bear anywhere within the purple pie on the ground plane results in a successful grasp, consistently. The areas are mirrored for the front right camera.

One exception to the area is when the object is too close to the edge of the pie directly ahead of Spot. When the object is placed in this area, the arm sometimes digs through the floor in an unpredictable motion.

Evaluation of stationary grasping

Although the area of object placements for a successful grasp is limited, the grasp success rate with the object in that area is 90%, or 9 out of the 10 grasps conducted in the area were successful. This is an important result, as it allows for high success rates when the object point is chosen within the area, and when the artificial object point is constructed to be within this area. The heading modification of $+0.2rad$ in the final pose of Spot (Turning Spot slightly towards the left) described in Subsection 4.2.5 increases the probability that the above conditions are met and improves the grasp success rate.

5.1.3 Accuracy of Spot and Kinova's poses

As a quadruped robot, Spot must be able to balance itself while walking to its destination. The balancing requirement sometimes causes the robot to end up at a slightly different end pose than is requested. The orientation is correct to $\pm.05rad$, whereas the position varies significantly. Testing Spot's movements was required to identify the reasons behind the errors in its final position.

What could cause the errors in the final pose?

A hypothesis about the reason behind the inaccuracy in Spot's movements is the moving arm changing Spot's centre of mass. The Kinova Gen3 arm weighs 8.2 kg, which is 21.6% of Spot's weight, mounted on its back. When the arm is fully extended and leans towards one side of the robot, it causes the robot to stumble towards the side the arm is extended towards. Sometimes the robot is able to correct this mistake and shuffle back to its path, but at other times it fails to do so.

How were the tests conducted?

To test the hypothesis, Spot and the teddy bear were positioned at known points. Then Spot was moved to the proximity point, and tape was placed in front of its front feet. Spot was then moved back to its previous position using the position history feature of `spot_mover` from Subsection 4.2.5, and then back to the proximity point.

This was repeated 10 times. Afterwards, the same action was repeated whilst the arm moved towards the teddy bear, to hover above it. The differences in Spot's position were recorded. An example from the tests is presented in Figure 5.6

Figure 5.6: Three examples of the Spot position accuracy tests, with a tape measure for reference. Reference tapes are encircled in red.

The data obtained is presented in Table 5.4.

Trial	Position relative to first try (cm)			
	Without arm movement		With arm movement	
	Left→right (− → +)	Back→front (− → +)	Left→right (− → +)	Back→front (− → +)
0	0	0	0	0
1	-5	+56	-65	+37
2	-16	+20	-47	+25
3	-10	+21	-28	+30
4	-5	+41	-3	+53
5	+10	0	-35	0
6	-3	-5	+20	-10
7	+11	-20	-15	0
8	-5	-10	-5	+17
9	-19	0	+13	-5
10	+7	-20	-24	+48

Table 5.4: Results of Spot movement accuracy experiment: Deviations in Spot's final position from trial 0

Evaluating test results

The data shows sporadic behaviour, with Spot stopping at unexpected positions at each trial, both with and without the arm. Therefore, the data disproves the hypothesis.

However, an observed pattern is that with the arm movement, Spot tends to drift more towards the left. This is because for the experiment, Spot's front right camera was used, which faces towards the left of Spot. Thus, the object is slightly towards the left of Spot, which leads to the arm generally following a leftward trajectory to reach the object, pushing spot towards this side.

This property shows that although Spot itself moves to unexpected positions, the arm movement also has significant effects on its final position. Furthermore, the arm did not follow a consistent trajectory - its movement was also unpredictable.

A major reason in Spot's erroneous end pose could be the imprecise object positions used to create a waypoint. However, it is evident that more factors contribute to the final pose error, as some results contain errors over 40 centimetres, which is more than the maximum error caused by the object detector uncertainties.

How about the arm's final pose?

The arm's final pose also carries significance. This is because the second grasp stage is conducted starting from the end pose of the first grasp stage. If the first grasp ends at an awkward pose for the arm, there might not be any kinematically feasible trajectories to the final grasp pose.

When the arm was instructed to move to the required position, the end effector position was always reliable, and did not vary like the final position of Spot.

The arm's trajectory

A major reason behind the arm's sporadic trajectory choice is thought to be the the selection of grasp pose generator. Small changes in the environment (such as a very small difference in Spot's initial pose) lead to large differences in the grasp pose generated by the spherical grasps server, due to the lack of an object pose. The grasp poses generated tend to require the arm to make large and unexpected movements, which lead to Spot stumbling to either side.

How could the problem be alleviated?

By estimating the object's pose and using it to generate the list of grasp poses, the arm's movements can be made more consistent. Furthermore, the grasp generator can also impose a final pose for the end-effector of the arm that will comfortably allow for trajectory planning for grasp stage 2.

The consistency in arm movements will remove one of the reasons for Spot's erroneous final position. Other factors affecting the position are likely to be the uncertainty in the object position and intrinsic properties of Spot's control system. However, the experiments showed that even though Spot moves to an erroneous position, it is aware of it - Spot's final position with respect to the vision odometry frame (as seen in RViz) is correct.

Therefore, planning a second trajectory to fix the mistake is an option to alleviate this problem. On the other hand, there is no guarantee that Spot's position after the second trajectory execution will be correct, as the factors causing the errors (imprecise object point and intrinsic controls) still exist during the second trajectory planning.

5.2 Overall System Results

The system components comprise the dynamic grasping pipeline. Therefore, limits in one of the components affects the functioning of the entire pipeline. This section discusses the effect of each component's performance on the success rate and performance of the achieved dynamic grasps.

5.2.1 Conditions Under Which the System Works

For a grasp to be successful, a number of conditions must be met. These are:

1. Grasp stage 1 and Spot movement
    (a) Object in view of Kinova vision module and detected successfully

(b) Generated artificial object point in allowable grasping area from current position

(c) Predictable and consistent motions of Spot and Kinova succeeding generated grasp pose and waypoint

2. Grasp stage 2

(a) Object in view of Spot vision module and detected successfully

(b) Object point in allowable grasping area

(c) There exist feasible trajectories from initial to final pose

Under the current circumstances, conditions 1.(c) and 2.(c) cause problems. There are two reasons why the problems exist. 1.(c) and 2.(c) are caused by the lack of object pose estimation, imprecise object points and Spot's intrinsic controls, as discussed in Subsection 5.1.3. It is expected that the introduction of object pose estimation in a future project will partially resolve these two issues and improve the reliability and performance of the system.

As detailed in Subsection 4.2.5, the artificial object point is calculated on the vector from Spot to the object. Condition 1.(b) requires that the artificial object point is modified to be in the allowable grasp area. To meet this condition, this point is moved towards the centre of the front left camera image. This solution fixes the motion planning failures stemming from wrong positioning of the artificial object point.

5.2.2 System Performance

To test the dynamic grasping pipeline in its entirety, 20 dynamic grasps were performed. Out of the 20 grasps, only 1 was successful. This is an extremely low 5% success rate.

The results demonstrate the unreliability of the current dynamic grasping system. Due to time constraints, potential solutions to unreliability issues were not implemented, resulting in the above mentioned low success rate.

In a future project, implementing object pose estimation and precise object location using multiple cameras could fix the inaccuracies and improve system reliability significantly.

Chapter 6

Discussions and Final Remarks

6.1 Evaluation

This section evaluates the final system (the product) in terms of its capabilities and compares the final version of the system to an ideal version of the same system (the ideal product). The ideal product would not have different features - it is simply the same system, but functioning without problems.

The product comprises a complete, dynamic grasping pipeline with Spot and Kinova that allows for object detection and grasping from a distance unreachable by moving only the manipulator arm. The pipeline is able to detect an object over a large distance and perform a two-stage grasp, where the first stage moves Spot and Kinova such that the gripper is hovering above the object, and the second stage performs the grasp from the adjusted intermediate state.

The ideal product would detect an object over a large distance and perform a two-stage grasp accurately and reliably, accurately moving the robot and arm at the same time and speeding up the grasping process even more. The success rate of the ideal product would be 100%, regardless of the initial position of the object.

For evaluation purposes, the product can be divided into three subsystems: Object Detection Subsystem, Grasping Subsystem, and Dynamic Grasping Subsystem. These three systems will be evaluated separately, with reference to the requirements specified in Section 1.2 and comparisons to their ideal versions. For the evaluation of each subsystem, results obtained in Chapter 5 will be used.

6.1.1 Object Detection Subsystem

The object detection subsystem is a combination of object detectors and an object localiser, and is capable of detecting objects around the robot and locating them in

the real world (In three-dimensional space). Both the object detectors and the object localiser have their strengths and weaknesses, which will be evaluated separately.

Object detectors

Implemented detectors

The object detector is a combination of a pre-trained and a custom-trained model. The pre-trained model is fast enough to detect objects in real-time in 24 FPS, and is good at detecting a large variety of objects (91 classes in COCO). However, due to the high number of object classes, it is prone to making mistakes when assigning detected objects to a class (For example, assigning a teddy bear to a fire hydrant).

The self-trained model is as fast as the pre-trained model, but is trained to only recognise one object. Unlike the pre-trained model, the limited number of object classes leads to close to 100% accuracy and reliability in detecting and, trivially, classifying the object if placed within 1.5 metres from the camera. The custom model's reliability at detecting the test object, in spite of the images being grayscale and fisheye, is its greatest strength. It is a proof of concept for using Spot's cameras for object detection tasks. In a future project, PRL can train an object detector to detect other classes with these cameras.

The object detector also includes a capability to seamlessly and quickly switch between the pre- and custom-trained models and input sources, which is an essential capability in making the two-stage grasp possible, and a strength of the system.

Ideal detector

The ideal product would make use of an external set of high resolution RGB-D cameras positioned around Spot, each with a field of view of around 90° (Wider than Kinova's but narrower than Spot's). These cameras would allow for more reliable object detection at a larger range of distances, all around the robot, as opposed to only in the arm's line of sight.

The higher quality of the cameras would also enable an object detection model to be trained on objects belonging to a larger set of classes, and still detect and classify each object very reliably. To train the model, a dataset would be created with relevant classes (For example, if the robot will be used to pick up trash, only objects commonly considered trash would be added).

Finally, the ideal object detector would also possess the ability to detect or estimate the orientation of an object.

Object localiser

The object localiser is capable of locating detected objects in three-dimensional space. Furthermore, the localiser can receive the images and bounding boxes from either source and seamlessly switch to the correct camera's intrinsics and depth

images, ensuring that there is never an error when locating an object. The calculated object position is always true to the depth image, but its accuracy depends on the accuracy of the depth data.

To improve the accuracy, the solution proposed in Subsection 5.1.1, using multiple cameras simultaneously to narrow down the object position, could be used.

In spite of the accuracy that could be improved, the object localiser has achieved all of its objectives. However, object localisation proved to be a very challenging task when switching to Spot's cameras, due to the BostonDynamics API's image encodings, extra processing and frame transforms described in Subsection 4.2.4.

Have the relevant requirements been fulfilled?

In spite of its flaws, the implemented object detection subsystem has achieved requirement 1 in Section 1.2, which required the product to detect and locate objects to grasp. However, the object detection is limited to one kind of object (A teddy bear), which is more limited than the ideal product. This could be improved by implementing the above mentioned changes. More changes to the subsystem are proposed in Subsection 6.2.2.

6.1.2 Grasping Subsystem

The grasping subsystem, upon receiving a three-dimensional object point, is able to grasp the object placed at the point using the Kinova arm. Its strength is its reliability when grasping objects placed within the allowable object area: If the object is placed in the correct area, then the grasp is successful over 90% of the time.

Its greatest weakness stems from the object detection subsystem not providing an object orientation. The lack of an estimate of the object orientation necessitates the use of the spherical grasps server for grasp pose generation, which does not require an object orientation. As a result, the arm performs unpredictable motions when grasping objects (Detailed explanation in Subsection 5.1.3).

The significance of this weakness was underestimated during the project design, and has led to problems with the concurrent motion of Spot and the arm. An ideal grasping system would, in addition to the subsystem's current capabilities, always grasp objects in a reliable, predictable motion. Additionally, it would be able to grasp rigid and irregularly shaped objects.

Have the relevant requirements been fulfilled?

Requirement 2 was achieved: The product is able to perform a stationary grasp for objects placed in a large area in front of the robot. However, the lack of pose

estimation led to unpredictable motions when grasping and made it very difficult to grasp irregular or rigid objects.

6.1.3 Dynamic Grasping Subsystem

The dynamic grasping subsystem is responsible for coordinating the motions of Spot and Kinova throughout the two stages of the grasp. During the first stage, Kinova and Spot are moved such that the gripper is within a few centimetres of the object. At the second stage, the arm quickly grasps the object.

The greatest strength of the subsystem is the two-stage setup which removes the requirement for a feedback system and still speeds up the grasp process compared to a sequential (static) grasp. Its greatest weakness is its unreliability when performing the motions: Spot usually ends up drifting from the path, which decreases the product's success rate to only around 5%. The reasons for this problem were explained in depth in Subsection 5.1.3.

An ideal system would be more precise in both motions, with the gripper always ending up hovering above the object, and the motion planning never failing in either stage of the grasp.

Have the requirements been fulfilled?

Requirement 3 has been achieved: A trajectory to move to the detected object's proximity is planned and executed successfully. Requirements 4 and 5 were partially achieved, since the implemented system is able to perform a dynamic grasp, but this grasp occurs in two stages rather than one, and the success rate is very low.

6.1.4 Evaluating System Performance

Considering the success rate of 5%, the developed system does not work as reliably as initially imagined. This is because of the unexpected roadblocks and challenges faced in the process, and could be improved in the future by implementing the improvements recommended in Subsection 6.2.2.

Requirement 6 states that the implemented solution must be able to perform a grasp faster than if robot movement and grasping were performed sequentially rather than concurrently. The second grasping stage was repeatedly tested, where the arm was initially only centimetres away from the object. About half of the successful grasps led to an improvement as opposed to grasping from the home position. The remaining half took approximately as long as sequential grasping, since the generated grasp poses caused the arm to move around the object unexpectedly and unnecessarily. Adding object pose estimation would also solve this problem, and allow for requirement 6 to be fully achieved.

6.2 Conclusion and Future Work

6.2.1 Conclusion

The project has managed to implement a product capable of performing a two-stage dynamic grasp with Spot and Kinova. The implemented system is also capable of performing a sequential grasp (First moving Spot, then Kinova).

Although the success rate of the dynamic grasp is low and the object choice is limited to a teddy bear, the project has served as a proof of concept. As such, it has made way for a future project in the PRL to implement a more reliable solution for dynamic grasping. Such a solution could have a much higher success rate, would be scaled up to grasp more objects at more varied locations around the robot, and perform a grasp in one step.

Such a project would not have to implement the pipeline from scratch, as a grasping pipeline is now already implemented - it would simply build on the current pipeline.

Additionally, the design and implementation process led to the creation of an extensive knowledge pool about technical implementation of some ROS packages, set-up of libraries and conversion of APIs to work with ROS. This knowledge pool was built up while the pipeline slowly took shape. The information gained is in the appendices, and will allow future projects in the topic to skip the set-up phases of libraries and speed up implementation.

Furthermore, by contributing a functioning pipeline of object detection and localisation using Spot's cameras rather than Kinova's (and a depth image rather than a computationally expensive pointcloud), the project has helped potential projects in PRL requiring object detection and exploration with Spot.

It is difficult to identify a single part of the project that could be considered the most difficult, since every idea led to many different problems, the solutions of which created even more problems - the problem complexity was $O(2^n)$.

A large setback in this project was not setting the description and scope clearly enough in the beginning. Future projects in the topic could set a clearly-defined scope and follow well-defined steps to reach it.

6.2.2 Future Work

As discussed in Section 6.1, the real product is lacking in performance when compared to the ideal product. The product could be improved: Made more reliable and capable to match the ideal product. Moreover, it is also possible to go beyond this

ideal version of the current system and instead perform a dynamic grasp in one step, without stopping at all - as listed in requirement 5 from Section 1.2.

Below, three technologies will be discussed. Implementing the first two will bring the product closer towards the ideal product, whilst implementing the third will move it towards performing a dynamic grasp in one step.

Object pose estimation

As part of the background theory, Subsection 2.2.2 introduced using a deep learning-based approach to estimate an object's pose, to use for grasp pose generation. As discussed in Subsection 5.1.3, integrating object pose estimation into the pipeline would improve consistency of arm movements, reducing unexpected motions and thus both improving the accuracy and execution time of grasps, and reducing the error in Spot's final pose caused by mistakes in Spot's centre of gravity caused by the arm's swings. Implementing this technology would be the first step towards improving the system's reliability.

Using external cameras for object detection

Using Spot's cameras for the final grasp is a good solution, since the cameras are fisheye, which makes it easy to locate the object when Spot is close to it. However, Kinova's vision module has a very narrow field of view, which makes it unsuitable for use in exploration tasks. Using multiple external RGB-D cameras stationed around Spot simultaneously to conduct object detection and the first stage of the grasp would enable Spot to perform exploration and significantly improve the capabilities of the system.

Furthermore, positioning the cameras in such a way that detected objects stay in sight from initial detection until the final grasp will remove the need for input source and object detector switching, enable the use of only one model to perform all stages of the object detection task and greatly improve the accuracy of the system.

Most importantly, the results in Chapter 5 showed that there are large uncertainties in the object position. Implementing a system where two cameras are simultaneously used to narrow down possible object positions and improve accuracy would allow objects to be located accurately from large distances, making possibly the greatest impact to the reliability and range of the system.

Such a system would work similar to GPS (Global Positioning System), where a device's location is determined by its distance to three near satellites with known location. In other words, such a system would be a scaled-up implementation of a stereo depth system.

In such a system, the depth value obtained from each camera would have an uncertainty of $\pm 10\%$, and the intersection of rays from the two cameras would be used to narrow the point down to a much smaller uncertainty.

A feedback system

Implementing a feedback system in addition to the aforementioned technologies will enable continuous correction of the object pose and motion plan, which will ultimately enable the grasp to be completed in one continuous action. Such a system would not require the motion planning to be repeated - instead, it would simply alter the existing trajectory of the arm by providing simple corrections.

As discussed in Subsection 5.1.1 and Subsection 5.1.3, a successful one-step dynamic grasp would require a feedback system to continuously correct the arm's and Spot's movements, in addition to an accurate and precise object position.

The interim report contained plans to implement a feedback system, listing two options:

1. The object detector and inverse kinematics solver can be re-run to calculate the path more accurately once every few frames,
2. A PID controller can be developed to calculate and correct deviations from the real path every few frames.

The first option proved infeasible due to the complexity. Running the object detector, localiser and inverse kinematics solver took several seconds, which made it impossible to run frequently enough to make the system more accurate. Moreover, as Spot starts moving, the object leaves Kinova's field of view, and enters Spot's field of view only as Spot stops moving. Thus, the object is not in view while Spot is moving, which makes it impossible for this method to work, unless external cameras are used, as described above.

The second method is more feasible, and could have been implemented. However, it does not solve the problem in its entirety, as the initial object point is not very accurate or precise, leading to large errors in the arm's goal state even if deviations are corrected with a PID controller. However, this method is expected to work if used in conjunction with external cameras or if the final object point is used to enhance it.

By implementing all three technologies, requirement 4 from Section 1.2 could be satisfied, allowing the system to perform a dynamic grasp in one stage, greatly reducing the time required to perform a grasp over a large distance.

Bibliography

Diar Abdlkarim, Valerio Ortenzi, Tommaso Pardi, Maija Filipovica, Alan Wing, Katherine J Kuchenbecker, and Massimiliano Di Luca. Prendosim: Proxy-hand-based robot grasp generator. In *ICINCO 2021*, 2021.

Andreas Aristidou and Joan Lasenby. Inverse kinematics: a review of existing techniques and introduction of a new fast iterative solver. 09 2009.

Bernard Bayle, Jean-Yves Fourquet, and M. Renaud. Kinematic modelling of wheeled mobile manipulators. volume 1, pages 69 – 74 vol.1, 10 2003. ISBN 0-7803-7736-2. doi: 10.1109/ROBOT.2003.1241575.

Patrick Beeson and Barrett Ames. Trac-ik: An open-source library for improved solving of generic inverse kinematics. In *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*, pages 928–935, 2015. doi: 10.1109/HUMANOIDS.2015.7363472.

C. Dario Bellicoso, Marko Bjelonic, Lorenz Wellhausen, Kai Holtmann, Fabian Günther, Marco Tranzatto, Péter Fankhauser, and Marco Hutter. Advances in real-world applications for legged robots. *Journal of Field Robotics*, 35(8): 1311–1326, 2018. doi: 10.1002/rob.21839.

C. Dario Bellicoso, Koen Kramer, Markus Stauble, Dhionis Sako, Fabian Jenelten, Marko Bjelonic, and Marco Hutter. Alma - articulated locomotion and manipulation for a torque-controllable robot. *2019 International Conference on Robotics and Automation (ICRA)*, 2019. doi: 10.1109/icra.2019.8794273.

Kondalarao Bhavanibhatla and Dilip Pratihar. Kinematic analysis of legged mobile manipulator. 12 2017.

BostonDynamics. Spot core specifications. https://www.bostondynamics.com/sites/default/files/inline-files/spot-core.pdf.

Chen Chen, Mengyuan Liu, Xiandong Meng, Wanpeng Xiao, and Qi Ju. Refinedetlite: A lightweight one-stage object detection framework for cpu-only devices. *CoRR*, abs/1911.08855, 2019.

Yu-Chen Chiu, Chi-Yi Tsai, Mind-Da Ruan, Guan-Yu Shen, and Tsu-Tian Lee. Mobilenet-ssdv2: An improved object detection model for embedded systems. In *2020 International Conference on System Science and Engineering (ICSSE)*, pages 1–5, 2020. doi: 10.1109/ICSSE50014.2020.9219319.

Marcus Gualtieri, Andreas ten Pas, Kate Saenko, and Robert Platt. High precision grasp pose detection in dense clutter, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

G. Heppner, T. Buettner, A. Roennau, and R. Dillmann. Versatile - high power gripper for a six legged walking. *Mobile Service Robotics*, 2014. doi: 10.1142/9789814623353_0054.

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

Albert S. Huang, Edwin Olson, and David C. Moore. Lcm: Lightweight communications and marshalling. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4057–4062, 2010. doi: 10.1109/IROS.2010.5649358.

Marco Hutter, Christian Gehring, Dominic Jud, Andreas Lauber, C. Dario Bellicoso, Vassilios Tsounis, Jemin Hwangbo, Karen Bodie, Peter Fankhauser, and Michael et al. Bloesch. Anymal - a highly mobile and dynamic quadrupedal robot. *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016. doi: 10.1109/iros.2016.7758092.

Mrinal Kalakrishnan, Sachin Chitta, Evangelos Theodorou, Peter Pastor, and Stefan Schaal. Stomp: Stochastic trajectory optimization for motion planning. In *2011 IEEE International Conference on Robotics and Automation*, pages 4569–4574, 2011. doi: 10.1109/ICRA.2011.5980280.

Kilian Kleeberger, Florian Roth, Richard Bormann, and Marco F. Huber. Automatic grasp pose generation for parallel jaw grippers, 2021.

Sulabh Kumra, Shirin Joshi, and Ferat Sahin. Antipodal robotic grasping using generative residual convolutional neural network, 2019.

Yuxi Li, Jiuwei Li, Weiyao Lin, and Jianguo Li. Tiny-dsod: Lightweight object detection for resource-restricted usages. *CoRR*, abs/1807.11013, 2018.

Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection, 2017.

Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single shot MultiBox detector. In *Computer Vision – ECCV 2016*, pages 21–37. Springer International Publishing, 2016. doi: 10.1007/978-3-319-46448-0_2.

Yugang Liu and Guangjun Liu. Kinematics and interaction analysis for tracked mobile manipulators. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 267–272, 2007. doi: 10.1109/IROS.2007.4399443.

Yugang Liu and Guangjun Liu. Mobile manipulation using tracks of a tracked mobile robot. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 948–953, 2009a. doi: 10.1109/IROS.2009.5354670.

Yugang Liu and Guangjun Liu. Modeling of tracked mobile manipulators with consideration of track–terrain and vehicle–manipulator interactions. *Robotics and Autonomous Systems*, 57(11):1065–1074, 2009b. ISSN 0921-8890. doi: https://doi.org/10.1016/j.robot.2009.07.007.

Yugang Liu and Guangjun Liu. Modeling of tracked mobile manipulators with consideration of track-terrain and vehicle-manipulator interactions. *Robot. Auton. Syst.*, 57(11):1065–1074, nov 2009c. ISSN 0921-8890. doi: 10.1016/j.robot.2009.07.007.

Xin Lu, Quanquan Li, Buyu Li, and Junjie Yan. Mimicdet: Bridging the gap between one-stage and two-stage object detection, 2020.

Arsalan Mousavian, Clemens Eppner, and Dieter Fox. 6-dof graspnet: Variational grasp generation for object manipulation, 2019.

MoveIt! Moveit deep grasps. https://ros-planning.github.io/moveit_tutorials/doc/moveit_deep_grasps/moveit_deep_grasps_tutorial.html, a.

MoveIt! Ros-planning/moveit_grasps: Geometric grasping generator library for cuboids. https://github.com/ros-planning/moveit_grasps, b.

MoveIt! Moveit! motion planners. https://moveit.ros.org/documentation/planners/, c.

MoveIt! Pilz industrial motion planner. https://ros-planning.github.io/moveit_tutorials/doc/pilz_industrial_motion_planner/pilz_industrial_motion_planner.html, d.

MoveIt! Difference between plans obtained by stomp, chomp amd ompl. http://docs.ros.org/en/kinetic/api/moveit_tutorials/html/doc/stomp_planner/stomp_planner_tutorial.html#difference-between-plans-obtained-by-stomp-chomp-and-ompl, e.

Jordi Pagès, Luca Marchionni, and Francesco Ferro. Tiago: the modular robot that adapts to different research needs. 2016.

Pal-Robotics. Pal-robotics/tiago_tutorials: Public tutorials of tiago robot. https://github.com/pal-robotics/tiago_tutorials.

Nathan Ratliff, Matt Zucker, J. Andrew Bagnell, and Siddhartha Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In *2009 IEEE International Conference on Robotics and Automation*, pages 489–494, 2009. doi: 10.1109/ROBOT.2009.5152817.

Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

Bilal Ur Rehman, Michele Focchi, Jinoh Lee, Houman Dallali, Darwin G. Caldwell, and Claudio Semini. Towards a multi-legged mobile manipulator. *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016. doi: 10.1109/icra.2016.7487545.

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. 2018. doi: 10.48550/ARXIV.1801.04381.

SBPL. Search-based planning library. http://www.sbpl.net.

C Semini, N G Tsagarakis, E Guglielmino, M Focchi, F Cannella, and D G Caldwell. Design of hyq – a hydraulically and electrically actuated quadruped robot. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, 225(6):831–849, 2011. doi: 10.1177/0959651811402275.

Sangok Seok, Albert Wang, Meng Yee Chuah, Dong Jin Hyun, Jongwoo Lee, David M. Otten, Jeffrey H. Lang, and Sangbae Kim. Design principles for energy-efficient legged locomotion and implementation on the mit cheetah robot. *IEEE/ASME Transactions on Mechatronics*, 20(3):1117–1129, 2015. doi: 10.1109/tmech.2014.2339013.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, December 2012. doi: 10.1109/MRA.2012.2205651. https://ompl.kavrakilab.org.

Changliang Sun, Yuanlong Yu, Huaping Liu, and Jason Gu. Robotic grasp detection using extreme learning machine. pages 1115–1120, 12 2015. doi: 10.1109/ROBIO.2015.7418921.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

Mingxing Tan, Ruoming Pang, and Quoc V. Le. Efficientdet: Scalable and efficient object detection. *CoRR*, abs/1911.09070, 2019.

Andrew Tao, Jon Barker, and Sriya Sarathy. Detectnet: Deep neural network for object detection in digits. https://developer.nvidia.com/blog/detectnet-deep-neural-network-object-detection-digits/.

Shantanu Thakar, Pradeep Rajendran, Ariyan M. Kabir, and Satyandra K. Gupta. Manipulator motion planning for part pickup and transport operations from a moving base. *IEEE Transactions on Automation Science and Engineering*, 19(1): 191–206, 2022. doi: 10.1109/TASE.2020.3020050.

Qijie Zhao, Tao Sheng, Yongtao Wang, Zhi Tang, Ying Chen, Ling Cai, and Haibin Ling. M2det: A single-shot object detector based on multi-level feature pyramid network, 2018.

Simon Zimmermann, Roi Poranne, and Stelian Coros. Go fetch! - dynamic grasps using boston dynamics spot with external robotic arm. *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021. doi: 10.1109/icra48506.2021.9561835.

Appendix A

Relevant Code

The project has contributed to two libraries. The first one is the `auto_grasp` ROS package, entirely written for the project. This library contains all nodes written for the project. The library can be found at the following GitHub link.

`https://github.com/Armanfidan/auto_grasp.git`

Furthermore, the Personal Robotics Laboratory's `spot_prl` package, which contains the Spot interface script, has been forked and modified. The fork can be found at the following link.

`https://github.com/Armanfidan/spot_prl.git`

Finally, all Docker images and Makefiles used in the project are contained in a fork of the PRL_Dockerfiles repository of Imperial College. This fork can be found at the following link.

`https://github.com/Armanfidan/PRL-Dockerfiles.git`

The relevant Dockerfile and Makefile are located in the "arman_spok" folder. Building this Docker image will restore the current state of the project's dependencies. For instructions on building the Docker images, refer to the README of the repository.

For access to any of the above private repositories, please contact the author directly.

Appendix B

Technical Details

B.1 Synchronising System Clocks with an NTP Server

In order to transform points between frames in ROS, a library called TF is used. TF constantly publishes frame transforms for each link of the robot, asynchronously. In order to ensure that all transforms correspond to the same point in time, TF allows transforming a point only if the frame transforms havce matching timestamps.

If the frame transforms are coming from different machines, the machines must have synchronised clocks for the frame transform timestamps for correspond to the same point in time. Thus, ROS master and all ROS nodes must be running an NTP (Network Time Protocol) server to ensure their clocks are synced. If this is not done, the header timestamps do not match up and this causes errors when trying to perform a transformation between links. The following link can be followed to run the reader through the setup for an NTP server and client.

`https://vitux.com/how-to-install-ntp-server-and-client-on-ubuntu/`

In the project, TF produced errors even after the clocks of both machines had been synchronised. This problem turned out to be because of the use of a simulator. The images were being received on a camera, whose timestamps were the milliseconds elapsed since 1970. On the other hand, the arm used was a simulation, whose system clock is the milliseconds elapsed since the simulator started.

A way to fix this is to either record the timestamp when the simulator starts and subtract this value from the camera timestamps, or to use the real arm rather than a simulation.

In the project, this was fixed by hotswapping the pointcloud message header timestamps

with the latest simulator clock instances. Finally, the script was told to wait for the latest transforms using the `tf` Python API.

B.2 Catkin Workspaces

A Catkin workspace is a versatile tool that allows developers to easily build their packages.

This section will briefly summarise practical facts about Catkin workspaces that may be useful for any future project working in the field.

A workspace has five spaces: source, build, install, devel and result. A workspace can be configured for development or release. If configured for release, the built binaries can be found in `$CATKIN_WS/install` (install space), and to access them, the `setup.bash` in this directory must be sourced. On the other hand, if the workspace is configured for development, package resources and commands are updated by sourcing `$CATKIN_WS/devel/setup.bash`.

Configuring a workspace for development gives the developer more debugging features, such as running code immediately after rebuilding, building components, and more.

Furthermore, when configuring a workspace, using `-cmake-args -DCMAKE_BUILD_TYPE=Release` enables compiler optimisations. Instead, `-DCMAKE_BUILD_TYPE=RelWithDebInfo` can be used for release optimisations with debugging symbols.

A space (such as devel or install) can be linked or merged. Merged means merging all build files in one large space, isolated means independent spaces for each package and linked means building each package into its own hidden tree and symbolically linking it into the unified devel space. The author has found that the merged devel space works best, and produces the fewest errors.

A workspace can be built with Catkin build tools or `catkin_make`. The two tools are incompatible, but Catkin build tools were used for the project, as it was found to work better, since it builds each package independently rather than in an intertwined way, which makes it more robust and produces fewer errors.

Note if using `moveit_grasps`

`moveit_grasps` was not building because of dependency incompatibilities. All dependencies should be built from the `melodic-devel` branch, with the exception of `rviz_visual_tools`, which should be built from the master branch. Additionally, moveit needs to be built from source, otherwise `moveit_grasps` installation will fail. To build moveit successfully, the workspace should be configured properly - the website lists an install configuration, but it is better to instead use a development setup.