Exploring Baxter Robot and Development of Python algorithms to Execute Holding, Lifting and Positioning Tasks

Rabé Andersson

2019

Student thesis, Advanced level (Master degree, two years), 30 HE
Electronics
Master Programme in Electronics/Automation

Supervisor: Sajid Rafique
Examiner: Per Mattsson
Exploring Baxter Robot and Development of Python algorithms to Execute Holding, Lifting and Positioning Tasks

Cover picture: Baxter Robot made by Rethink Robotics is holding a cube, Rabé Andersson, June 2019
Preface

First and foremost, I would like to thank my supervisor Dr Sajid Rafique for his excellent guidance and support in writing this thesis. I also wish to thank all the respondents; without whose cooperation, I would not have been able to conduct this work.

To my teachers at the University of Gävle: I would like to thank you for your wonderful cooperation. It was always helpful to bat ideas about my research around with you. I have also benefitted from debating issues with my friends and classmates.

Special thanks to my wonderful wife Sarwin for her selfless love, care and dedicated efforts which contributed a lot for completion of my thesis.

I owe thanks to my beloved parents for their continued and unfailing love, support and understanding during my life and my pursuit of writing my thesis. I consider myself the luckiest in the world to have such a lovely and caring family, standing beside me with their love and unconditional support.

I thank the Almighty for giving me the strength and patience to work through all these years so that today I can stand proudly with my head held high.

Finally, I would like to dedicate this work to my parents, wife and family.

Without you all, I could not have achieved my goals.
Abstract

The greatest feature of using a Baxter robot among other industrial robots is the ability to train this robot conveniently. The training of the robot could be done within a few minutes and it does not need so much knowledge of programming. However, this type of training feature is limited in functionality and needs frequent updating of the software and the license from the manufactural company.

As the manufacturer of Baxter Robot no longer exists due to a merger, the thesis has twofold aims, (1) Exploring different functional, installation, calibration, troubleshooting and hardware features of the Baxter robot and (2) demonstrate the functionality of Baxter to perform general tasks of holding, lifting and moving of test objects from one desired position to another position using custom-made programs in Python. Owing to this, details about different software and hardware aspects of Baxter robot is presented in this thesis. Additionally, systematic laboratory tutorials are also presented in appendices for students who want to learn and operate the robot from simple to complicated tasks. In order to keep the Baxter operational for students and researchers in future, when there is no more help available from its manufacturer, this thesis endeavour to cover all these aspects. Thus, the thesis presents a brief understanding of how to use the Baxter Robot in a simple and efficient way to perform a basic industrial task.

The kinematics part will show the concepts of forward and inverse kinematics and the DH (the Denavit–Hartenberg) parameters that are important to understand the end-effector position according to the world frame that will give the knowledge of those who are interested in the kinematics part of Baxter robot.

The work of the thesis will make it easier to understand how to program a Baxter robot by using Python language and using the simplest way to move the arm to certain positions.

The ROS principles, kinematics and Python language programs will provide a good platform to understand the usability of Baxter robot. Furthermore, easy to use laboratory tutorials are devised and are presented in the appendices. These laboratory tutorials will improve the understanding of the readers and provide a step-by-step guide of operating Baxter robot according to the principles of Robotics.

In addition to all these points above, the thesis shows useful functions that are built in ROS (Robot Operating System) that make it easier to program the robot in an untraditional way which is one of a contribution of this thesis itself.

The usual way to program the robots, in general, is to study the robot kinematics and calculate the position of the end-effector or the tool according to some frame or the world coordinate frame. This calculation can be done by the forward kinematics or the inverse kinematics. The set of programming Baxter robot in this thesis is not the complex calculation of the forward or the inverse kinematics. The tf (transform) tool in ROS has made it easier to reach the joint angles and program Baxter robot using Python.
# Table of contents

1 **Introduction** ................................................................. 1  
   1.1 Aims and Objectives of the thesis ........................................ 2  
   1.2 Contributions of the Work .................................................. 2  
   1.3 Structure of the Thesis ..................................................... 3  
2 **Theory** ........................................................................ 4  
   2.1 Baxter Robot ....................................................................... 4  
      2.1.1 Baxter Collaborative Robot (CoBot) ................................... 4  
      2.1.2 Hardware Description ..................................................... 5  
      2.1.3 Prominent Features of Baxter Robot .................................. 8  
      2.1.4 The desired environment ................................................. 9  
   2.2 Robot Operating System (ROS) ............................................. 12  
      2.2.1 Robot Operating System (ROS) ....................................... 12  
      2.2.2 Why ROS? .................................................................. 13  
      2.2.3 Computation Graph and the file system ............................. 13  
      2.2.4 Roscore ..................................................................... 14  
      2.2.5 ROS workspace and Packages ...................................... 16  
   2.3 Kinematics of Baxter robot .................................................. 17  
      2.3.1 DH parameters ............................................................ 18  
      2.3.2 Forward Kinematics ...................................................... 26  
      2.3.3 Inverse Kinematics ....................................................... 26  
      2.3.4 tf (Transform Library) .................................................. 27  
3 **Process and results** ...................................................... 28  
   3.1 ROS configurations ............................................................ 28  
   3.2 Changing the display image to Baxter’s head ......................... 29  
   3.3 Python Code .................................................................. 30  
   3.4 Result ........................................................................... 33  
4 **Discussion** ..................................................................... 35  
   4.1 Difficulties that faced programming Baxter robot ................... 36  
5 **Conclusions** ................................................................... 38  
   5.1 Future Work ................................................................. 39  
References ............................................................................. 41  
Appendix A: Laboratory Tutorials ............................................... A1  
Appendix B: Python Code ....................................................... A41  
Appendix C: Hardware specifications ......................................... A50
LIST OF ACRONYMS

FSM: Field Service Menu
PSF: Python Software Foundation
ROS: Robot Operating System
PNG: Portable Network Graphic
GIF: Graphic Interchange Format
IDE: Integrated Development Environment
API: Application Programming Interface
SDK: Software Development Kit
SSH: Secure Shell
DNS: Domain Name System
NTP: Network Time Protocol
LTS: Long Time Support
USB: Universal Serial Bus
PyPI : Python Package Index
tf: Transform library
IK: Inverse Kinematics
EOL: End Of Life
LIST OF FIGURES

FIGURE 1: BAXTER ROBOT AND THE BAXTER HEAD [4]
FIGURE 2: BAXTER HARDWARE FRONTVIEW [6]
FIGURE 3: BAXTER HARDWARE BACKVIEW [6]
FIGURE 4: CONDITION RING OF BAXTER ROBOT AND THE LIGHT MEANINGS [6]
FIGURE 5: CONDITION RING OF BAXTER ROBOT [6]
FIGURE 6: BAXTER SCREEN (FACE) EXPRESSION [7]
FIGURE 7: BAXTER SCREEN FOR THE BAXTER RESEARCH ROBOT [8]
FIGURE 8: BAXTER GRIPPER [9]
FIGURE 9: GRIPPER OF BAXTER'S CUFF [10]
FIGURE 10: USING THE TRAINING CUFFS [6]
FIGURE 11: USING THE TRAINING CUFFS [6]
FIGURE 15: EXAMPLE OF MESSAGE COMMUNICATION [18, p.59]
FIGURE 16: THE MESSAGE COMMUNICATION IN BAXTER ROBOT
FIGURE 17: BAXTER ARM AND JOINTS [21]
FIGURE 18: BAXTER ROBOT WITH RANGE OF MOTION - BEND JOINTS [21]
FIGURE 19: BAXTER RANGE OF MOTION - TWIST JOINTS [21]
FIGURE 20: BAXTER LEFT ARM AND ITS LINK LENGTHS [21]
FIGURE 21: SEVEN-DOF RIGHT ARM KINEMATIC DIAGRAM WITH COORDINATE FRAMES [3, p.8]
FIGURE 22: BAXTER'S BASE COORDINATE SYSTEM [22, p.192]
FIGURE 23: BAXTER LEFT- AND RIGHT-ARM KINEMATIC DIAGRAMS, TOP VIEW ZERO JOINT ANGLES SHOWN [3, p.10]
FIGURE 24: TOP VIEW DETAILS [3, p.10]
FIGURE 25: Baxter Front View [3, p.11]
FIGURE 26: BAXTER HEAD SCREEN IMAGE THAT IS USED IN THIS THESIS
FIGURE 27: THE CUBES ARE ARRANGED IN (4*3) DIMENSIONAL ARRAY TO THE LEFT WITH THE POSITION P1-P12 AND THE POSITIONS T1-T12 THAT ARE ARRANGED IN (3*4) DIMENSIONAL ARRAY TO THE RIGHT
FIGURE 28: DIMENSION OF THE CUBE THAT ARE USED AND THE DISTANCE BETWEEN THE POSITIONS IN mm
LIST OF TABLES

TABLE 1: JOINT RANGE TABLE (BEND JOINTS) [21]

TABLE 2: JOINT RANGE TABLE (TWIST JOINTS) [21]

TABLE 3: LENGTHS OF BAXTER LEFT ARM [22]

TABLE 4: SEVEN-DOF RIGHT ARM DH PARAMETERS [3, p.8]

TABLE 5 SEVEN-DOF LEFT- AND RIGHT-ARM JOINT LIMITS [3, p.9]

TABLE 6: {B} TO {WO} LENGTHS
1 Introduction

We live in the age of increasing robot use in our daily life. There are many different types of robots in the industries, among them is an industrial robot which is built by Rethink Robotics and is founded by Rodney Brooks. It is called a Baxter robot [1] and is explored in this thesis as shown in Figure 1.

![Baxter Robot and The Baxter Head](image)

**FIGURE 1: BAXTER ROBOT AND THE BAXTER HEAD [4]**

This thesis shows the principles and procedures that have been used in the Baxter robot to perform some functions such as lifting, holding and moving an object by robot’s gripper then leaving that object in a specific place.

The main aim in this thesis is to study the Baxter robot and to understand the robotics principles such as ROS (Robot Operating System) and the kinematics which makes it possible to program the robot by using Python language and to enlarge the scope of vision of robotics in general. The programming language, which is used in this thesis, is Python language. Python language is preferred in many robotics applications [2, p.96].

It is worth mentioning that most Baxter robots which are used in the industries are made to perform functions such as holding, lifting and positioning of an object. These functions enable Baxter robot to be efficient both in time and money in the factories’ lines where the Baxter robot is easy to be used and mounted in different stations for achieving various tasks. Baxter robot can be trained by the worker or the user in a swift and easy way (This is illustrated in laboratory tutorial 3, p.A31), but this feature has its limitation. This leads the thesis focusing on how to program Baxter robot by using Python language.
1.1 Aims and Objectives of the thesis

The manufacturing version of a Baxter robot can be easily trained by holding the cuff to enter G-zero mode that lets the arm to move freely and moves the arm to a desired position and by using the navigator buttons either in the arm or on the torso of the Baxter robot to make the robot remembering this position and coordinates. This is a distinct feature that Rethink Robotics presented in Baxter robot unlike to other industrial robots. This manufacturing version of Baxter robot has its limitation and depends on the preprogrammed features that Rethink robotics has made for its customers. In addition to this limitation, the manufacturing version needs frequent software updating, license and FTP account \[26, p. 187\][23]

The other version that Baxter robot offers is the Baxter research version. The University of Gävle has the Baxter research version that is why it has been used in this work. The following main objectives:

1. To study and install the software that works with Baxter robot and to understand the hardware functionality and requirements of the Baxter Robot.
2. To study and implement principles of ROS (Robot Operating System) of Baxter Robot.
3. To investigate the kinematics of Baxter robot (the forward kinematics; the Inverse kinematics) and study the DH (the Denavit–Hartenberg) parameters.
4. To test the robot with tasks such as holding, lifting and positioning the test-piece objects (cubes) using two techniques, (1) the manufactured provided training mode in the Field Service Menu (FSM) and (2) using own developed programs written in Python language to perform holding, lifting and positioning tasks.
5. To develop smart and systematic laboratory tutorials for learners and students.

Finally, the work of the thesis also aims to support the sustainable development vision of the University to improve technological engagement and creativities in teaching and research. The work will increase knowledge and awareness of prospective students and researchers about the challenges in the adoption of robots and CoBot to achieve sustainable growth in the rapidly growing area of automation.

1.2 Contributions of the Work

1. Using Python language in writing the custom-made programs to achieve holding, lifting and positioning of objects (12 cubes) and moving them from one arrangement array (4*3) into another place to be arranged in an array (3*4) dimension.
2. The set of programming the Baxter robot in this thesis uses ‘tf’ (transform) library. The ‘tf’ library is used to reach the actuator’s status and to read the 7-joints angle values. By reading the angle values of each joint and using Python dictionary makes it easy to program Baxter robot.
1.3 Structure of the Thesis

This thesis work consists of five chapters:

Chapter 1: Briefly presents an introduction to the topic and objectives, contributions and structure of the thesis

Chapter 2- Theory: Details about Baxter Robot components, Robot Operating System ROS concepts, the message communication in ROS, kinematics part of Baxter robot and the reason of choosing Python.

Chapter 3- Process and result: describes step by step method in ROS configuration and Python scripting and the result in the real Baxter robot.

Chapter 4- Discussion: Discuss the difficulties of programming Python and the advantage of this work in the robotics field.

Chapter 5- Conclusion: provides the overall summary of this thesis and the recommended future work.
2 Theory

This section provides an overview of the Baxter Robot and robotics in general as well as illustrates benefits of using a Baxter robot. It can help the user to figure out which environment and the software are needed before beginning the journey in the field of Baxter robot.

Robot Operating System (ROS) concepts will be illustrated and how the message communication occurs in ROS and a brief understanding of the reason for choosing ROS and Python in this work is presented.

2.1 Baxter Robot

Baxter robot was introduced in September 2011 by Rethink Robotics Inc. and founded by an Australian roboticist Rodney Brooks [1]. Before embarking into the details of the Baxter robot, it is pertinent to introduce the idea of a collaborative robot which is also called coBot.

2.1.1 Baxter Collaborative Robot (CoBot)

Baxter is termed as a coBot as it is designed to directly work with human beings. The coBot is intended to physically interact with humans in a shared workspace. Baxter is designed to be unique and untraditional from the other industrial robots by its safe interaction with humans. The Baxter robot consists of two arms with dual 7-degree-of-freedom (dof) and 2-dof head with an animated face. It has a vision system that is represented by three cameras. One camera is on the head and two cameras are on the arms: one camera is on each arm. The robot has a control system, a safety system and a collision detection sensor [3, p. 3].

Baxter is 3 feet tall and weighs (74.84kg) without its pedestal; with its pedestal, it is between 5'10" – 6'3" tall and weighs (138.79kg). See Figure 1 (p.1) that shows the Baxter robot with the pedestal and Baxter head.

It is used for simple industrial jobs such as loading, unloading, sorting, and handling of materials. Baxter is designed to perform dull tasks on a production line with a great feature that can be trained and programmed easily [1].

The animated screen of the robot "face" allows to display multiple facial expressions. These facial expressions are used in the manufactural version to interact with the workers [4].

There are sets of sensors on its head that are used to sense people nearby and give Baxter the ability to adapt to its environment. That is one of the great features of Baxter that gives the ability to adapt to its surroundings, unlike other industrial robots which should be caged. Baxter also has extra sensors in its hands that allow it to pay very close attention to detail. Baxter runs on the open-source Robot Operating System on a regular, personal computer which is in its chest.

Finally, Baxter can also be placed on a four-legged pedestal with wheels to become mobile [1].
2.1.2 **Hardware Description**

The hardware description of the Baxter robot is shown in Figure 2 and Figure 3 that illustrate the main parts in the Baxter hardware.

**Condition Ring**

The condition ring around the head gives the condition of the robot. Figure 4 shows the position of the ring and the light colour meanings.

![Condition Ring Diagram](image)

**FIGURE 4: CONDITION RING OF BAXTER ROBOT AND THE LIGHT MEANINGS [6]**
Attention Ring

A cluster of two or three yellow lights will appear towards the object movements when Baxter detects any movements within its workspace. Figure 5 shows the position of the attention ring.

![Attention Ring Image](image)

*FIGURE 5: CONDITION RING OF BAXTER ROBOT [6]*

Baxter facial expressions

The industrial Baxter robot has different facial expression and it is good for Baxter user to know these expressions meanings. Figure 6 shows all these expressions.

![Baxter Facial Expressions Image](image)

*FIGURE 6: BAXTER SCREEN (FACE) EXPRESSION [7]*

The screen of the Baxter Research version can be seen in Figure 7 and how to change the screen to another image is illustrated on page 29 and in Appendix A (Laboratory tutorial 3, p.A38).

![Baxter Screen Image](image)

*FIGURE 7: BAXTER SCREEN FOR THE BAXTER RESEARCH ROBOT [8]*
Baxter Gripper

Baxter has two grippers: one gripper is located in each arm that can be seen in Figure 8.

Electric Parallel Grippers

Vacuum (Suction) Grippers

*FIGURE 8: BAXTER GRIPPER [9]*

The Baxter’s cuff

Baxter has cuff in each arm, and they are important in moving the arm and pick and release objects. See Figure 9 and Figure 10.

*FIGURE 9: GRIPPER OF BAXTER’S CUFF [10]*

1. Training cuff switch
2. Grasp button
3. Action button

*FIGURE 10: USING THE TRAINING CUFFS [6]*
The Navigator

The navigator is on each arm which is used to scroll to and interact with options on the screen. There are two additional navigators on the torso, one navigator on each side of the torso. When you press the OK button (2) (or the action button on the cuff), the white indicators on the navigator light up. Figure 11 is showing the Navigator that is located on the arms and on the torso.

![Navigator Diagram](image)

**FIGURE 11: USING THE TRAINING CUFFS [6]**

2.1.3 Prominent Features of Baxter Robot

Many industrial robots are made to perform one or a few tasks with very fast movements which make these robots unsafe for the human to work beside. This is a reason that many robots need to be caged and, in many cases, they stop working if something enters into their workspace. The case is different with the Baxter robot. The Baxter robot is even supported by a screen that can also have different facial expressions to interact with the workers [1].

Baxter Robot has become a part of the workforce which challenges with the humans in a full world of technology and rapid development. It has many features that make it better than many other industrial robots. These features are [12]:

- It is safe to work next to the people in the production line without the need for caging like the other traditional industrial robots: KUKA, ABB Montoman and Mitsubishi etc.
- Baxter robot has an easily integrated feature that can be easily connected to other automation lines without the third-party integration.
- It has force detection that makes the robot react for different environments.
- Baxter can work in different stations in the production line and it is easy to install it wherever it is needed and that makes it cheaper in a comparison with other industrial robots. Any updating in any automation line of traditional industrial robots eats up time and financial output in any case of changing [5].
• Despite the great feature that Baxter robot has that can be trained to save money and time in its operation, but this training feature is still limited in functions. Programming Baxter using Python or C++ is the solution for this limitation and that is what will be illustrated later in other chapters.

2.1.4 The desired environment

Before beginning to work with Baxter, it is very important to know the desired environment for the Baxter robot. This environment means both the hardware and the software required for Baxter robot to work within.

• The hardware environment

The desired hardware environment for the Baxter robot is the workspace that Baxter moves in without having any obstacles. This hardware environment is the dimension (space) that is required for Baxter in order to work freely for all type of its applications [11]. The information about the dimension of Baxter robot is also important in the calibration of Baxter’s arms and the springs (in case needed) since the robot needs to do a full stretching of the arms in the calibration operation. During the calibration operation, no obstacle should be in Baxter workspace. The Figures (12-14) show the Baxter workspace from a different point of view and this information is needed in case of the calibration of the arms and the springs.


• The software environment

The software environment for Baxter to work within this thesis is Ubuntu 14.04 (which is recommended by Rethink Robotics) which lets (ROS) Indigo to be installed and to communicate with Baxter [13]. The installation of ROS Indigo can be found in Appendix A while the programming language that is commonly used for Baxter robot is Python language or C++. The programming language which is used in this work is Python.

Python language

Python programming language is a high-level and general-purpose programming language. It is released in 1991. It is used in many functional programming, object-oriented paradigms and can be classified as a structured programming language. Python’s developer is the Python Software Foundation (PSF) which is devoted to the Python language in order to develop the Python distribution, managing intellectual rights and develop conferences [24].

Many operating systems include Python programming language and the Python interpreter as a standard component. Python programming language has many various versions. Python versions according to their dates of release have been started with Python 0.9.0 that is released on February 20, 1991 and the up-to-date version Python 3.7 that is released on June 27, 2018[25]. The Python version that is used with the Baxter robot in this work is Python 2.7.6.

Why Python?

Python is an object-oriented programming language that is used in writing scripts whether for a machine or a websites design. Python differs from other programming languages like C or C++ in the execution of the code since Python can execute the same task but with fewer lines [26].

The main idea of using Python programming language in this work compared to the other programming language that Python is commonly used in artificial intelligence, embedded systems and Robotics with the help of its good libraries. These libraries make the programming of machine learning and Artificial intelligence much more helpful compared to other programming languages. The official repository for third-party Python software which is called Python Package Index (PyPI), contains over 130,000 libraries. These libraries are used in many different areas like: Graphical user interfaces, Web frameworks, Multimedia, Scientific computing, Image processing etc. [27].

Furthermore, Python has high information security in the industry and most robots are programmed in Python language. To move the Baxter robot and start understanding the ROS concepts, it needs to deal with an easy programming language. It is readable and the syntax forming is easy to understand and easy for the beginners as the professional programmers in expressing their ideas and concepts in a very readable way to follow up the code by using a few lines.
2.2 Robot Operating System (ROS)

When we talk about a series of a personal operating system so “Windows” comes as an answer directly. The same way as the Robot Operating System (ROS) that works in the field of Robotics [13, p.9].

The details about Robot Operating System (ROS) and the Software Development Kit (SDK) that is used with Baxter and how to install them, are illustrated in Appendix A. There are several types of ROS versions but the recommended ROS version by Rethink Robotics is ROS Indigo that works well with the recommended Ubuntu system (14.04).

In this chapter, some ROS concepts will be illustrated to build a knowledge of the steps that are done in the thesis and will also present the overview of the ROS which will help to understand the laboratory tutorials which are attached in the appendices of the thesis.

For the better understanding of the reader, the following aspects and their relationship will be covered in this chapter to explain ROS:

- What is ROS?
- Why ROS?
- What are the computation graph and the file system?
- Creating ROS workspace and Packages and what are they?
- Understanding ROS topics.
- Communication in ROS.

2.2.1 Robot Operating System (ROS)

ROS is an open-source system for any robot. It provides the services that are expected from an operating system which includes low-level device control. It implements different functions, communications between processes and package management [14].

ROS was initially released in 2007. ROS is a community-driven project that led by the Open Source Robotics Foundation (OSRF) [13, p.16].

ROS is completely free and customizable which means that everyone can customize the ROS according to the requirements of their robots. All other components can be removed, and a specific platform can be customized to the behaviour of the robot application.

The hardware abstraction layer can be provided by ROS as this makes it easier for the developer and the robot application programmer to build their own applications without thinking much about the hardware aspects. Thanks for having ROS in the robotics field that makes different types of software tools which can be used to easily control or visualize important data of the robot. [13, p.16]
The software in ROS is organized in packages which are the heart of all communication in any robot. These packages consist of small folders or nodes. These nodes are small executable programs that are created to achieve some tasks of the robot functions. The packages can also contain processes, datasets and configuration files that are organized in a single set or module [13, p.18]

2.2.2 Why ROS?

The fact is that there were many robotics’ researches before ROS that work on specific robots. It was hard to understand and follow up by other researchers or programmers. The solution is to build a generic software which enables researchers and universities to reuse tools or libraries. If a company or a group of researchers are good in image processing and others in the robotic manipulator, so both can reuse their packages and libraries which save the time for developing the aspect in the robotic field. This is called the collaborative development of the ROS [13, p16].

ROS has become a generic framework to program all kinds of robots and many companies like: Fetch Robotics, PAL Robotics, DJI, and Robotics are mainly using ROS in their works [13, p.37-38].

Other reasons for choosing ROS are:

- For the language support: ROS can deal with different programming languages like C++ and Python that make it easy to exchange codes and packages between researcher.

- For the library’s integration: ROS can use different libraries that can be called whenever the programmer needs them. Some of these useful libraries are: rospy, Open Source Computation Vision (Open-CV) can be called by using `import cv2` or to import the ROS Python library that can be called by `import rospy`. This is what will be explained later through the Python code in the Process chapter.

- For the simulation integration: ROS is an open-source that uses also the open simulator such as Gazebo that can also be a useful tool to test the code that is written in Python or C++.ROS has also an inbuilt testing tool such as `rostest` that can check the code [13, p. 16]

2.2.3 Computation Graph and the file system

Any robot has numbers of sensors, actuators and computing components that are connected all together through the robot control system and Baxter robot has no exceptions. These sensors and actuators motivated the design of ROS to fulfil the feature “fetch an item” problem. In the “fetch an item” term, the robot’s task is to navigate an object in the workspace or environment, finding the object and placing it in the requested location [17, p.9].

Baxter robot has a series of Elastic Actuators (SEA), manipulator arms, motorized gripper, camera, infrared sensors and accelerometer on the cuff at the end of each arm as well as 360-degree sonar sensors at the top of Baxter’s head. All these sensors make the robot react to the environment and their works need to be structured [15].
Having the sequence of tasks in the robotics need to structure the software and to divide the control software into a low-level independent loop. These low-level independent loops connect the nodes by high-level supervisor quarries or language to manage them in building the robot program. The supervisor quarries do not need to know the implementation details of any of the low-level control loops since it communicates with each node only by using simple control messages. The nodes communicate with each other by edges. This is what makes the system modular and easy to reuse the same code across many robotics platforms.

ROS Computation Graph builds a set of software easily and the communication between these control loops occur through the node within the computation graph. The node is a simple and executable file that performs some tasks and should be located under the package. Furthermore, the nodes can be represented like small codes that achieve a small task and can be written in different programming languages such as Python or C++. The nodes can be dependent on other nodes, libraries or nodes that are written in a different programming language. All these spread nodes can be organized in ROS under the package to manage these nodes and libraries. This is what ROS can provide as a file system concept [14].

All the communication between these nodes occurs when the nodes find each other but what happens if the network is busy? how do nodes find each other and how does the communication start? The program roscore has the answer to this dilemma [17, p.11].

2.2.4 Roscore

The nodes communicate with each other by using the message. The links between these nodes are called edges which represent the actuators commands, sensor data and joints position states. All these nodes are connected with the heart of ROS that is called roscore. The roscore is a service or a program that has all the connection information between the nodes within ROS. It is responsible for identifying all different nodes and will have all the information about the list and topics [18]. It can be activated by using the command roscore.

An example that shows the connection between the nodes and the master node: roscore is that connection between the node turtlesim and the keyboard with the help of the master node roscore.

The keyboard node will act as a publisher node that will publish a velocity command and the turtlesim node that receive and subscribe these commands from the keyboard. Finally, it executes the command to make the robot move in the simulated environment.

The procedure of this communication can be done as follows and shown in Figure 15:

1. The node turtlesim will send a request to the master node roscore in order to declare itself into the master node. It tells the master node about the name of the node and the topic name which is /turtle1/cmd_vel and the type of the message is geometry_msgs/Twist. Twist message is sending the linear speed and the angular speed in ROS.
2. The publisher node (keyboard node) will repeat the same procedure as node 1 and it declares itself in the master node which is in this case /teleop_turtle and the topic name /turtle1/cmd_vel and the message name that is geometry_msgs/Twist. When this information has reached to the master node roscore that the both nodes matching in the topic name and the message type.

3. The master node will then send the information to the node turtlesim that tells it about the publisher node which is called /teleop_turtle and which has a message name: geometry_msgs/Twist. The master node will ask the /turtlesim to contact the /teleop_turtle that fulfils its behaviour.

4. The turtlesim node will send a request to /teleop_turtle and establish a TCPROS communication and /teleop_turtle will send a response TCPROS connection protocol. The connection will be established and each command that comes from the keyboard will move the robot /turtle in the simulation.

![Diagram](image)

**FIGURE 15: EXAMPLE OF MESSAGE COMMUNICATION [18, p.59]**

The same procedure happens when we run example 4 in Appendix A (laboratory tutorial 3, p.A21) to control the Baxter robot by using the keyboard.

Controlling the Baxter robot via Python code can be achieved as follows:

The roscore will be activated as soon a Python code is running. This will let the joint_position node in Baxter robot to send its information to the roscore in ROS telling roscore about the node name, topic and the message type which will be of the type cmd_vel (command velocity). By starting the Python code, it will do the same procedure with the Master node roscore.
When all the information about the nodes is identified in the Master node roscore, the Master node sends the information to the joint_position node telling it that there is a node which fulfils the messages that the joint_position is seeking. As a reaction for the joint_position node will send a TCPROS request to the Python code control node which will respond with TCPROS response. As a final step, the communication will be established between the Python code node and the joint_position node in the robot. Each time the code is sending the command to Baxter robot, a message of the type /cmd_vel will be sent to the joint position and command the robot to move Baxter’s arm.

The same procedure will be repeated with Python code for holding, lifting and positioning the cubes by using Baxter robot. The communication is illustrated in Figure 16:

![Message Communication in Baxter Robot](image)

*FIGURE 16: THE MESSAGE COMMUNICATION IN BAXTER ROBOT*

### 2.2.5 ROS workspace and Packages

It can be difficult for the ROS beginner to distinguish between the ROS workspace and ROS package and whether there is some relationship between them. A simple way to think about the workspace is an environment or the set of directories for the Robot Operating System (ROS) to work with the code which can be written in Python, C++ or any other programming language.

ROS can have many workspaces but only one of them can be activated and worked with at a certain time. The creation of a workspace in ROS can be started by deciding where the workspace will be in the operating system. In many literature books as well as in the ROS website(www.ros.org), the workspace is called catkin_workspace catkin_ws *(The ROS user can also call it anything).*

The workspace includes other subfolders which are: build, devel and src.
• \textit{build} is a folder inside the ROS workspace where some libraries and programs are going to be stored. It is not usually convenient that the ROS user cares much about this folder.

• \textit{devel} is a development folder that contains several files and dictionaries. The most important one is the setup files.

• \textit{src} is the source file that contains all the packages that include the codes which are written to achieve some task in the robotics field [17, p.13-14].

All these concepts are needed to understand the environment that facilitates Baxter’s work.

Other terms that are used frequently in the ROS field is the package. Each project or code must be stored in a package. A package is a folder that is located inside the source folder \textit{src} that is inside the workspace. These packages contain the codes that are written in Python programming language, C++ or any other programming language. The packages contain other documentation and files that are needed to run the code. These important files are \textit{CMakeList.txt} file and \textit{package.xml}. The \textit{package.xml} describes the contents of the package and how the workspace will interact with it.

All the codes and the programs that achieve some task using Baxter robot or any other robots will be inside the source file \textit{src} inside the package. More details and examples can be seen in Appendix A (laboratory tutorial 2, p.A11).

2.3 Kinematics of Baxter robot

A proper understanding of the kinematics of any robot assists the operator or a programmer to come closer to develop the understanding of the robot workspace. This is one of the reasons to discuss the kinematics of the Baxter robot in this work. Another reason is to study the positions and the motions that a Baxter robot can achieve and to understand the limitations of these motions [19].

The study of kinematics for Baxter robot will measure the kinematic quantities used to describe the motion of points, objects and the joints position of Baxter robot without considering the forces that cause these motions. The kinematics, in general, is used to calculate the motion, velocity, acceleration and displacement. The most interesting calculation in kinematics, is the joint motion, velocity and the displacement of Baxter’s arms.

In order to describe the motion of any joint in Baxter robot, the position of the joint must be described in terms of a relative reference frame. This concept will be used in finding the DH (Denavit–Hartenberg) parameters that are used to calculate the position and the orientation of the end effector according to the reference frame and it will give the whole properties of the Baxter robot. Combining the DH parameters with either the forward kinematics or the inverse kinematics which is used to find the joint angles, the position and the orientation of the end effector which is the gripper in Baxter case [20].
2.3.1 DH parameters

Baxter robot has two arms. Each arm has seven joints that control the motion of the arm. So, each arm has \((7\text{-dof})\) and \((2\text{-dof})\) for the head which leads that Baxter robot has \((16\text{ dof})\). The seven joints and their names according to Rethink Robotics are shown in Figure 17.

![Baxter Robot Joints](image)

**FIGURE 17: BAXTER ARM AND JOINTS [21]**

S0: Shoulder Roll  
S1: Shoulder Pitch  
E0: Elbow Roll  
E1: Elbow Pitch  
W0: Wrist Roll  
W1: Wrist Pitch  
W2: Wrist Roll

These 7 rotary joints are be classified into two groups:

1. The bend or pitch joints. They are S1, E1, and W1.

The reason for calling them pitch joints is because they move up and down on the arm and rotate about their axis perpendicular to the joint. See Figure 18 that shows these joints and Table 1 with the range.
Joint Range Table (Bend Joints)

<table>
<thead>
<tr>
<th>Joint</th>
<th>(Degrees) Min limit</th>
<th>Max limit</th>
<th>Range</th>
<th>(Radians) Min limit</th>
<th>Max limit</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>-123</td>
<td>+60</td>
<td>183</td>
<td>-2.147</td>
<td>+1.047</td>
<td>3.194</td>
</tr>
<tr>
<td>E1</td>
<td>-2.864</td>
<td>+150</td>
<td>153</td>
<td>-0.05</td>
<td>+2.618</td>
<td>2.67</td>
</tr>
<tr>
<td>W1</td>
<td>-90</td>
<td>+120</td>
<td>210</td>
<td>-1.5707</td>
<td>+2.094</td>
<td>3.6647</td>
</tr>
</tbody>
</table>

TABLE 1: JOINT RANGE TABLE (BEND JOINTS) [21]

2. twist or roll joints. They are S0, E0, W0, and W2.

They are called roll joints because they rotate about an axis that sticks out their centerline.

The joints as S0, S1, E0, E1, W0, W1, and W2 will have their limitations also as it is shown in the following table. Figure 19 and Table 2 show these joints and its values.
The range of motion for each joint shows the measurements in degrees and radians.

The angles and measurements that ROS deals with are in radians [22, p.189]. These different joint values will expand the knowledge of the possible value that the joints can have. It will be used in the Python code later to achieve holding, lifting and positioning task.

In robotics, we are concerned with the position of the end effector in three-dimensional space.

This end effector is linked with other links of the robot that has different length as shown in Table 3 and Figure 20.
FIGURE 20: BAXTER LEFT ARM AND ITS LINK LENGTHS [21]

<table>
<thead>
<tr>
<th>Length</th>
<th>Value (mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_0$</td>
<td>270.35</td>
</tr>
<tr>
<td>$L_1$</td>
<td>69.00</td>
</tr>
<tr>
<td>$L_2$</td>
<td>364.35</td>
</tr>
<tr>
<td>$L_3$</td>
<td>69.00</td>
</tr>
<tr>
<td>$L_4$</td>
<td>374.29</td>
</tr>
<tr>
<td>$L_5$</td>
<td>10.00</td>
</tr>
<tr>
<td>$L_6$</td>
<td>368.30</td>
</tr>
</tbody>
</table>

TABLE 3: LENGTHS OF BAXTER LEFT ARM [22]

Link lengths for Baxter’s joints are measured in mm, from the centre of one joint to the centre of the next [22].

We notice that $L_6$ in Table 3 is not the same as what is shown in Figure 20. The reason for that is $L_6$ in Table 3 is taking into account the distance from the wrist pitch centre to the centre of the parallel-jaw gripper finger so (138.77 mm + 229.53 mm) = 368.30 mm.

In general, to find the position and orientation of a body in space, we need to attach a coordinate system, or frame to the object. So, management of coordinate frames plays an important role in robotics.

In very simple words, the position is a vector of three numbers ($x$, $y$, $z$) that describes the translation of any point of the object that is on the robot according to the $x$-axis, $y$-axis and $z$-axis of the free space.
The orientation is also a vector with three numbers (roll, pitch, yaw) that also describes how much the point or the object that belong to the robot is rotated with respect to the axis. All these calculations are according to one origin frame that can be defined in the beginning of the calculation [22, p.236].

The information about rotation and position can be placed in a single matrix called homogeneous matrix that will be used frequently to transform the information of any point or object in the robot according to some reference coordinate frame.

The homogenous Matrix can be calculated from the DH parameter (they are four values or parameters) that describe the robot property.

Each robot has its own DH parameters according to the number of frames, joints and their type if the joints are “Revolute” joints or “Prismatic” joints. It depends also on the links length that form a robot. For Baxter robot, the DH parameters are described in Table 4 and Figure 21.

![FIGURE 21: SEVEN-DOF RIGHT ARM KINEMATIC DIAGRAM WITH COORDINATE FRAMES [3, p.8]](image)

Due to the design symmetry of the Baxter robot’s arms, the DH parameters for the right arm is the same as the left arm.

In order to find the DH parameters of Baxter robot, following notations shall be noted:

\[ \alpha_{i-1} \]: rotation around \( X_n \) from \( Z_{n-1} \) to \( Z_n \)

\[ a_{i-1} \]: displacement along \( X_n \)

\[ d_i \]: displacement along \( Z_{n-1} \)

\[ \theta_i \]: rotation around \( Z_{n-1} \)
TABLE 4: SEVEN-DOF RIGHT ARM DH PARAMETERS [3, p.8]

These parameters in Table 4 will be used then in the Homogenous matrix.

Each robot has its limitation in joints according to its design and the Baxter robot is a good example. The joint angle limits for the Baxter robot with 7-dof are shown in Table 5 below, where the angles here are in degree.

<table>
<thead>
<tr>
<th>Joint Name</th>
<th>Joint Variable</th>
<th>( \theta_{i \text{ min}} )</th>
<th>( \theta_{i \text{ max}} )</th>
<th>( \theta_{i \text{ range}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_0 )</td>
<td>( \theta_1 )</td>
<td>+51°</td>
<td>-141°</td>
<td>192°</td>
</tr>
<tr>
<td>( S_1 )</td>
<td>( \theta_2 )</td>
<td>+60°</td>
<td>-123°</td>
<td>183°</td>
</tr>
<tr>
<td>( E_0 )</td>
<td>( \theta_3 )</td>
<td>+173°</td>
<td>-173°</td>
<td>346°</td>
</tr>
<tr>
<td>( E_1 )</td>
<td>( \theta_4 )</td>
<td>+150°</td>
<td>-3°</td>
<td>153°</td>
</tr>
<tr>
<td>( W_9 )</td>
<td>( \theta_5 )</td>
<td>+175°</td>
<td>-90°</td>
<td>265°</td>
</tr>
<tr>
<td>( W_1 )</td>
<td>( \theta_6 )</td>
<td>+120°</td>
<td>-175°</td>
<td>350°</td>
</tr>
<tr>
<td>( W_2 )</td>
<td>( \theta_7 )</td>
<td>+175°</td>
<td>-175°</td>
<td>350°</td>
</tr>
</tbody>
</table>

TABLE 5 SEVEN-DOF LEFT- AND RIGHT-ARM JOINT LIMITS [3, p.9]

Before writing the homogenous transformation matrix for both arms, we need to define a base coordinate system that will be in the centre of the torso. This base coordinate system will be used as a reference for all measurements of the arms in the Baxter robot. The Figure 22 below shows that the X-axis is going forward from the torso centerline and Y-axis is to the left from the centerline of the torso and finally, the z-axis is perpendicular to the torso centerline.
More details about the joints’ angles and from the top view can be seen in Figures (23-24).
From Figures (23-24), we can see that the length offsets $L$ and $h$ for the joint centres of each $S_0$ are shown in Table 6:

<table>
<thead>
<tr>
<th>Length</th>
<th>Value (mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>278</td>
</tr>
<tr>
<td>$h$</td>
<td>64</td>
</tr>
<tr>
<td>$H$</td>
<td>1104</td>
</tr>
</tbody>
</table>

Table 6: $\{B\}$ TO $\{WO\}$ LENGTHS
Finally, an example of the homogenous transformation matrix for both arms and after looking at Figure 23 and by substituting in H, L and h, the result will be as follows:

\[
\begin{bmatrix}
\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & -L \\
\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & -h \\
0 & 0 & 1 & H \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

2.3.2 Forward Kinematics

After studying the DH parameters of Baxter arm and each arm in the Baxter robot has 7-dof that means seven joints, so each arm has seven angle values to define the rotation of these joints. Since the length of these arms are fixed and as it is shown in (Table 4, p.23) and the joints angles are also known with their limitation (see Table 5, p.23). So, the calculation of the position and orientations of the gripper will be easy to find. The most common approach is to use the DH parameter where each joint is characterized by four parameters in the DH parameters table. These DH parameters are multiplied together to find the end effector (gripper) position and orientation according to one reference coordinate frame [19].

The calculation of the position and orientation of the end effector (gripper) from the joints angle values to be used as inputs to the transformation equation in kinematics is called the forward kinematics solution. Using this method will give only one solution for the end effector position since the joints angle values are set to one position or values.

One way to find these values of the seven joints angles can be accessed and monitored through a ROS tool which is called tf (short for transform). Another way to find the seven joints angle for each arm is by reaching the joint status topic.

2.3.3 Inverse Kinematics

In order to command a robot arm to some specific position in the robot workspace, it needs to know the end effector position and the calculation of the joints angle values. It will be found by using the transformation equation that is based on the DH parameters of a robot. An example of how to program Baxter robot by using the inverse kinematics method is illustrated in Appendix A (Laboratory tutorial 3, p.A21).
We have seen how these coordinate frames are a big deal in robotics in general and especially in forward and inverse kinematics. The calculation for these coordinate frames and the transformation can take a time but with the tf library that ROS provides, it becomes much easier to manage these coordinate frames. This method is used in this work [22].

2.3.4 tf (Transform Library)

tf is a tool that is extremely widely used throughout ROS software. It is used to keep track of the robot coordinate frames over time and it can work in a distributed system. The ability to work with the distributed system means all the coordinate frames of any robot can be monitored by ROS user which uses this tool or library.

The principle of managing these coordinate frames is achieved by nodes that can publish the information about some transform(s) in one place which make sense to be easier to use or monitor these information by the user of any other nodes in ROS. All the information of the coordinate frames is with respect to time and a node that has an access to a joint encoder to read the joint data is a great feature for this tool[17, p.27]. The method of reading the angles of all Baxter joints is used and will be illustrated later in (3. Process and result, p.32). The way of reading the joints angle values (for all 7 joints) will be shown in the process part.
3 Process and results

As discussed before in the theory part, in order to achieve the set tasks of holding, lifting and positioning using Baxter robot, it can be done either by the training method which does not need to be involved in writing any code or using the skill in Python language that is used in this work to program Baxter robot.

The default training mode method can be done by entering the Field Service Menu FSM and the way of investigating this method can be found in Appendix A (Laboratory tutorial 3, p.A25-A34) and the result can be followed in the result part (p.33). Due to the reason that the FSM is limited, and it is based on recording and playing back the movements to achieve a task one time without having the ability to repeat the sequence hence the second method which is through writing Python program is used.

So, the second way for achieving the task of holding, lifting and positioning an object is by programming the Baxter robot. A few configurations must be done before writing the code in Python language. Writing the Python code will be impossible without these configurations. These steps are recommended to be followed in case of writing the Python code to achieve any task when using Baxter.

3.1 ROS configurations

After installing the ROS Indigo on the Ubuntu 14.04 that is connected to Baxter robot so some initial ROS configuration should be done to enable the ROS and the Ubuntu operating system to establish communication with the robot and to prepare the step of writing the Python code.

ROS workspace represents the working area of the robot. Any workspace should have special files. These special files are setup.bash and the setup.sh. They must be executed in order to enable the workspace and establish the communication with the robot. The Baxter workspace contains all the files that are needed to run ROS and ROS application.

In the first step and in order to run the setup.bash and setup.sh every time the communication is established with the robot so it should be run in the terminal window. The procedure is not practical to write the commands and run these files manually in the terminal window by writing the commands to run setup.bash and setup.sh. Instead, these files will be added to the file (.bashrc). The file .bashrc contains some scripts about the workspace and it is executed automatically every time the user will open the terminal window. In order to run the setup.bash and the setup.sh which are important in communication within ROS, the following lines are added to the file .bashrc:

```
source /opt/ros/indigo/setup.bash
source /home/hig_student/Thesis/baxter_ws/devel/setup.bash
echo $ ROS_PACKAGE_PATH
```
Since the file .bashrc will be executed each time we open new terminal so the first line that has been added will tell Ubuntu to run the ROS configuration. While the second line tells where the software packages will be. Furthermore, the package will contain the Python code and the node. The Python code or any other code whether is written in Python or C++ should be located under the package directory.

The last line shows the package path at the beginning of each terminal window as a text. This guides the package that ROS will search this path each time to run the code.

Another important step in the ROS configuration is that working directly on a default workspace is not a practical way that the ROS users usually do. The reason for that is that any changes or problem with the default workspace will cause a problem with the ROS workspace [16].

So, I create a workspace which is called Thesis which contains the baxter_ws. Baxter workspace has all the packages that contain the nodes. These packages will represent all the projects while each node inside a package will represent the code. The package used inside baxter_ws is called (picknplace).

After creating the package (picknplace) and compiling the workspace, the Python code Baxter_works.py should be inside the src of the package (picknplace).

### 3.2 Changing the display image to Baxter’s head

Since the industrial software “INTERA” that is used with Baxter robot, has many facial expressions and the Baxter research version does not have any face expression so I changed the display image into painted human face because Baxter robot is a humanoid robot. Another reason for having the facial expression is to imitate the real manufacturing version and to interact with the people that are looking to the robot. The image that is used with the Baxter robot must be in the png or jpg extension.[28].

“A PNG file is an image file stored in the Portable Network Graphic (PNG) format. It contains a bitmap of indexed colors and uses lossless compression, like a .GIF file but without copyright limitations.”[29]

The way that shows how to change the display image (the research display image) into another image, can be seen in Appendix A (Laboratory tutorial 3, p.A38). In this thesis, the display image to Baxter’s head is called (happyface.png) and the result of the new display image in the head is shown in Figure 26:

![Figure 26: Baxter Head Screen Image That is Used in This Thesis](image-url)
3.3 Python Code

Writing a python code for holding, lifting 12 cubes that are arranged in (4*3) array and positioning them in a new array with (3*4) dimension as illustrated in Figure 27.

Where the cubes that are used are identical in size and each cube has the dimension (25mm*25mm*25mm) and the distance between the cubes can be seen in Figure 28.

Scripting task for achieving: holding, lifting and positioning cubes using Python language has been started by holding the cubes that are arranged in (4*3 array) on the left to the array (3*4) on the right as shown in Figure 27 above. This starts by holding the cube P1 and lifting it to P1Up which is 20 cm above the position P1 then the Baxter arm will move to the position T1Up that is also 20 cm above the position T1 and finally, the arm moves to T1. This will be repeated for all the cubes that are in P2-P12 that will end with the new positions T2-T12.
To sum up the picking and placing objects by using Python code with one of Baxter’s arms (the right arm, the procedures need to be structured in small tasks. This is what happens with the most robotics’ abstraction.

- The first step is that the position of picking the object(cubes) should be determined. The position will be in the space according to the world coordinate frame. The idea is to give the angles of all the 7-joints that the arm has.

- The second step is to command the arm to move toward the object’s position. Many control loops (one in each revolute joint) should be involved. These control loops are the position, velocity and controlling the power on these revolute joints which consequently let the arm move to the desired position.

- Third, is to command the gripper to open and close to grasp the object and hold it. The controlling loop in this step will be to receive the command to open and close the gripper. The motorized gripper should be on and Off.

- The last step is to move the arm to the new position that the object will be placed in. In this case, the cubes will be rearranged to the new array (3*4) dimension. The desired position will also be predetermined and based on the angles of all the revolute joints in the 7-dof arm. The sequence will be repeated for all cubes until they will be moved and placed in the new positions.

The program that is written in Python is repeating the sequence in an infinite loop that builds the “Automatic operation of Pick and Place” and the finite number of sequence repetition to build the “Manual operation of Pick and place the cubes”. This is also a distinct characteristic of the Baxter robot that the user does not have to manually measure the coordinate of the pick and place of the test objects. Instead of manually moving the gripper to these locations and reading the 7-joints angle values which can be called by using the dictionary feature in the controller of Baxter robot.

Finally, there are many points that are combined and used in programming Baxter robot using Python language:

1. As it is discussed in the kinematics part that the difference between the forward kinematics and the inverse kinematics is that the forward kinematics is used to find the position of the end effector by knowing the angles of the joints for the robot arm. While the inverse kinematics is used to calculate the angles of the joints by giving the end effector position. Other important difference is that the forward kinematics can have only one possible solution since the joint angles are fixed (These are the inputs of our equations) while the inverse kinematics can have many solutions since the same end effector position can have different joint angles solutions.

2. It is good to mention that Baxter robot has a good feature comparing with other industrial robots such as KUKA, ABB, Montomon etc. is that Baxter arm and the end effector position can be easily moved to the desired position if we hold the cuff of the robot that makes the arm to enter the G-zero mode and to make the arm to move freely.
3. Since there are four modes of controlling Baxter's arms [22, p.193]:

- **Joint velocity**: the message that can be read in ROS in this mode contains seven velocity values for the controllers to achieve. **This mode is dangerous, and its use is not advised.**

- **Joint torque**: The message that can be read in ROS in this mode contains seven torque values for the controllers to achieve. **This mode is dangerous, and its use is not advised.**

- **Raw joint position**: This mode provides a more direct position control, leaving the execution of the joint commands primarily for the controllers. **This mode is dangerous, and its use is not advised.**

- **Joint position**: This mode is the primary method for controlling Baxter's arms. The angle of each of Baxter's joints is specified in the message to the motor controllers. This message contains seven values, one value for each of the seven joints and thanks to the tf in ROS that is easy to access to the joint encoder data. This method is the key for this code scripts in this thesis.

The inverse kinematics is not used in this work since the joints angle values can have many solutions that lead to have different joints movement. However, the way that is used instead is to monitor the seven joints angle values for the arm that is used (right here) and store the values in order to ensure that each arm will take one solution of the joint angle values in its movement. Another reason is to avoid collusion in case of having any obstacle in the arms workspace while the robot achieves its tasks. It means having more control on the arm movements. In other words, it is to use the ability to move the Baxter arm freely to any position by entering the G-zero mode which is absent in many industrial robots as well as programming Baxter robot is not using the home point that will be a reference for the other coordinate positions.

Combining all this information together in one method and by using the tf ability to read Baxter seven joints (as illustrated in tf, p.27) values of each arm. The joint position control mode is implemented in this work and in order to read the seven-joints values for each arm, the following code is used:

```python
import rospy
import baxter_interface

rospy.init_node('Reading angles')
limb = baxter_interface.Limb('right')
anglesR = limb.joint_angles()
print anglesR
```

By printing each seven joints values, these values can be saved and put them in a dictionary to represent P1 (for the first position). This position P1 will be used again to command the robot arm to move the end effector to that position where the seven joints values are represented in P1.
It is similar in some sense the forward kinematics and the inverse kinematics principles. Firstly, it seems the inverse kinematics principle by giving the tf library, the end effector position and from the tf ability, the joints values will be calculated and viewed to be saved in a Python dictionary.

Secondly, the forward kinematics principle in some sense, is by giving the seven joints values and the tf will calculate the end effector position. By this position, the robot arm moves to these values(positions) that lead the robot end effector to the same positions that we chose first to represent P1.

This procedure will be repeated for all positions P1-P12 as well as the positions P1Up-P12Up that are 20 cm above the respective positions.

The same idea is used with T1-T12 and T1Up-T12Up that are also 20 cm above the respective positions.

- The positions P1-P12, P1Up-P12Up, T1-T12 and T1Up-T12 Up are be stored in Python dictionary.

**Running the Python Code**

After writing the Python code in a name Baxter_works.py and save it in the computer with the path:

/Thesis/baxter_ws/src/picknplace/Baxter_works.py

So, it is the time to run the code and to see the result. The details of how to execute the Python code can be followed in Appendix A (Laboratory tutorial 3, p.A39).

**3.4 Result**

Since the Baxter research version is used in this work so two methods are presented to execute the tasks: holding, lifting and positioning of cubes. The first method is achieved by entering the Field Service Menu FSM. The result of this method can be seen in this link:

https://www.youtube.com/watch?v=t5NsP-J295s

One of the disadvantages of the first method is the impossible duty to repeat the sequence because this method is based on recording the positions and playing the sequence that let Baxter robot move from one position to another.

The solution for this disadvantage is the way of having more control on the Baxter arms and achieving the tasks: holding, lifting and positioning the cubes by using Python programming language.
The second method of achieving the tasks which are holding, lifting and positioning can be seen by the following links:

**Part 1: Holding, lifting and positioning 12 cubes written in Python language by using Baxter robot.**

This part will operate of holding, lifting 12 cubes that are arranged in an array (4*3) dimension and positioning them in a new place and arranged in another array (3*4) dimension.

It shows the manual running mode and it can be seen in this link:

https://www.youtube.com/watch?v=ciX_I5FGFrU&t=54s

**Part 2: Holding, lifting and positioning 12 cubes written in Python language by using Baxter robot.**

This part shows the holding and lifting the 12 cubes that are arranged in (3*4) dimension and positioning them in the array (4*3) dimension. It is backwards operation to the first part.

It is showing the manual running mode as we. See the link below:

https://www.youtube.com/watch?v=ZOt9jLJS8_Y&t=14s

**Part 3: Holding, lifting and positioning 12 cubes by using Baxter robot, monitor the positions**

This part is the same manual running mode that are shown and illustrated in part 1 and part 2 above but this part is focusing on the messages and information that appear on the screen while running the Python program. The messages and the information can be seen in the link:

https://www.youtube.com/watch?v=HkNseuQ__85g

**NOTE:**

The Automatically running mode will do the same operation as is illustrated in part 1 and part 2 but the only difference is that Baxter robot will continue moving the cubes(12 cubes) from the first place(4*3) dimensional array to (3*4) dimensional array then it moves them back to the first position that is arranged in (4*3) array. It will continuously repeat these tasks unless CTRL+C will be pressed to stop executing the program.
4 Discussion

A Baxter robot that is placed in the laboratory of ATM at the University of Gävle is used to achieve this research. The aim of this research is to give a brief understanding on how to achieve a sequence of tasks: beginning with holding, lifting and finally simple positioning of an object in some predefined places that the user or the student defines.

To achieve these tasks which are also used widely in many industrial production lines by different robots, Python programming language has been used and (identical cubes) are used as objects to be moved. The reason for using the Python language is illustrated in this work as well.

To operate holding, lifting and positioning tasks, the objects that are used to be moved by Baxter robot, are cubes: The operation is achieved in two techniques,

1. Using the training mode in the FSM Field Service Menu and the objects are (5 identical cubes). Due to the limitation in this technique and missing ability to repeat a sequence of operations more than one time, so the second technique is used to program Baxter by Python language.

2. Using a developed program that is written in Python programming language to perform holding, lifting and positioning tasks. The objects are (12 identical cubes). These cubes are arranged in an array (4*3) that will be moved into another place and to be arranged in an array (3*4) dimension.

The reason for using the joint position control to do these tasks have been discussed. The most usual way to program the robots, in general, is to study the robot kinematics and calculate the position of the end-effector according to some frame or the world coordinate frame. This thesis covers how this calculation can be done either by using forward kinematics or inverse kinematics. Another important way that is mentioned in programming robots, in general, is to define the home coordinate point that is used as a reference for the other points in robot’s workspace (especially when the IK is used in programming) but the set of programming Baxter robot in this thesis is not the complex calculation of the forward or the inverse kinematics. Since the ROS has tf tool that makes it easier to reach the joint angles and by saving their status and call them back in a dictionary concept. It is the idea that is been used in the Python code here.

Many advantages can be summarized throughout the thesis:

1. Baxter robot enlarges the vision in the robotics field as well as attaching the main problems that many engineers can have in the industries.

2. Understand the Robotics principles by implementing the principles of ROS (Robot Operating System) of Baxter Robot and using the Ubuntu operating system in the computer that deals with Baxter robot.

3. To study, install the software that works with Baxter robot and to understand the hardware functionality and requirements of the Baxter Robot.
4. To study the DH (the Denavit–Hartenberg) parameters of Baxter robot that has 7 dof in each arm and to investigate the kinematics understanding that the Robotics lectures presented in the Automation program at the University of Gävle.

5. It discusses the communication between the nodes and the knowledge about messages in ROS.

6. It shows how it is important to have all the documentation and contacts with the manufacturing team support of any robot.

7. It gives a practical problem that enlarges Python vision, the skill in using Python and see how the dictionary feature in Python is useful.

8. To develop smart and systematic laboratory tutorials for learners and students.

9. The training features are based on artificial intelligence and approved by the Baxter robot behaviour in this project.

4.1 Difficulties that faced programming Baxter robot

Each work has its challenges which appear in functioning and programming any robots. The Baxter robot has some of these specifications and difficulties:

- The calibration of the electric gripper. The problem appeared when programming the electric gripper to hold the cubes, an error message came in each time the Python code had been run. The message was” the command cannot be done before calibration”. While sending the same command to the suction gripper, the suction gripper was working well without having this error message.

Many methods have been tried to calibrate the arms and the grippers, but they were useless. The absence of web information was also a barrier that made it difficult but fortunately, the problem was solved, and the solution of the calibration problem can be done by:

```python
#!/usr/bin/env python
import rospy
from baxter_interface import gripper as robot_gripper

#-------------------Initializing the right gripper-------------------
rospy.init_node('gripper_test')
#Set up the right gripper
right_gripper = robot_gripper.Gripper('right')
#------------------- Calibrate the right gripper-------------------
right_gripper.calibrate()
rospy.sleep(2.0)
```
print('Calibrating the right gripper now...')
#----------------------- Close the right gripper -----------------------
print('Closing the right gripper...')
right_gripper.close()
rospy.sleep(2.0)
#----------------------- Open the right gripper -----------------------
print('Opening the right gripper...')
right_gripper.open()
rospy.sleep(2.0)
print('Calibration is Done!')
5 Conclusions

In this work, a real robot called Baxter, manufactured by Rethink Robotics Corporation, has been described. However, due to a merger of ‘Rethink Robotics’, the online help is not really available. Therefore, this thesis explored and presented effectively different functional, installation, calibration, troubleshooting and hardware features of the Baxter robot. Furthermore, details about different software and hardware aspects of the Baxter robot is presented in this thesis. Additionally, systematic laboratory tutorials are also presented in appendices for students who want to learn and operate the robot in future to execute simple to complicated tasks. In order to keep the Baxter operational for students and researchers in future, when there is no more direct help available from its manufacturer, this thesis endeavour to cover all these aspects and in itself is a contribution.

**The second chapter** where the theory part gives a brief understanding of the Baxter components. Baxter's arms, sensors and the control modes for the arms are described. This understanding is followed by looking deeply into the requirements for Baxter environment both the hardware and the software requirements. The hardware requirement is represented by the space that is needed for the Baxter robot to have in order to let Baxter work freely in its workspace. While the software requirements are represented with expressing the required version of Ubuntu as well as the reason for having ROS in Robotics.

The main principles used in ROS and Baxter have been discussed which make the work possible. Furthermore, the differences between the workspace and packages are presented which cannot be obvious for the ROS beginners to distinguish between them. The communication between the nodes, the subscriber nodes and the publisher nodes have been shown and how all communications occur while the heart of ROS which is called ROS master or roscore is activated.

**The third chapter** two techniques to achieve the tasks (holding, lifting and positioning any objects) have been introduced:

- The first one is by using the training concept that is a great feature in Baxter robot. This training technique is based on (AI) Artificial Intelligence which is a big deal in Baxter world. The training method can be done by entering the FSM. How to achieve this technique is shown in detail in the laboratory tutorial 3 and the result can be seen on www.youtube.com, see the result part (p. 33).

Since Baxter has two movable arms, the joint position control is used to program the right arm of the robot.

- The second technique to achieve the tasks is by using a Python program. The programming is based on using the facility of moving Baxter arm to any position, read the seven joints angle values (by reading the joints status) and save them under some names. P(1-12), PLup(1-12), T(1-12) and TUp (1-12) names, joint position control and the dictionary principle in Python code are used. The result can be seen in the result part (p. 34).
It is worth mentioning that before beginning to program Baxter robot using Python language, ROS practical configurations must be done to create a working environment. Finally, the robot can be discovered more by going through 3 laboratory tutorials in Appendix A.

After installing ROS Indigo and making some required configurations which are illustrated in the first laboratory tutorial, ROS configurations and tutorials are followed to demonstrate various concepts of ROS and more different examples which are supplied by Rethink Robotics of controlling and testing Baxter ability are covered in the last laboratory tutorial.

5.1 Future Work

This thesis focuses on a brief understanding of holding, lifting and positioning task without using the image processing and the cameras that are mounted on Baxter’s arms. The recommended future works are:

- One of the greatest features of Baxter robot is visual servoing control that allows to detect and grasp objects. The cuff camera and gripper combination make it possible to determine and grasp objects. 2D camera images that are provided by Baxter's cuff camera has an important task to achieve this procedure. It can be processed by computer vision software such as OpenCV. OpenCV provides a vast library of functions for processing real-time image data. Many concepts of Image processing and Computer vision algorithms for thresholding, shape recognition, feature detection, edge detection, and many more are useful for 2D (and 3D) perception.

- When using the camera and the 2D camera images, the lighting issue is a big problem especially when working with the camera calibration. Focusing on camera calibration and how to use the finely tuned methods that deal with the calibration in surrounding lightning effect is a great work that is recommended to implement by using Baxter robot.

- A project that also focuses on trajectory and motion planning with the use of different grippers both with the physical robot and the simulated one by using Gazebo and Moveit.

- The simulated Baxter can be run using Gazebo. This environment will only show the Baxter robot. A project that simulates different real industrial environments with different objects will expand the opportunity for studying Baxter robot in different simulated environments. Creating a Graphical User Interface (GUI) that allows the user to click an object in the virtual world in Gazebo, and pick and place that particular object will also implement the opportunity to control and program the Baxter robot remotely that gives the students who are reading Robotics and are not on the campus) to program and control the robot (online).

- Connect the Baxter robot with the KUKA robot that is available in the laboratory of the Electronic Department and achieves some functions which can mimic the real station in some production line in the industries. It would enlarge the knowledge of both the theoretical and the practical part in the Robotics course.
• A project that is using a new versions of Ubuntu and ROS since the version of Ubuntu which is used in this thesis is 14.04 and for ROS version is Indigo.
References


Appendix A: Laboratory Tutorials

Laboratory tutorial 1: ROS and SDK installation

The targets of this laboratory tutorial are:

- Install the Linux operating system Ubuntu 14.04 which is recommended to work with Robot Operating System (ROS) indigo for Baxter robot.
- Install the Robot Operating System Indigo (ROS-Indigo).
- Install the Source Development Kit (SDK) for Baxter.

NOTE

The ROS can be installed in two ways: either by using the binary installation or by using the source compilation. The source compilation will be created as an executable by compiling ROS sources code which will take more time and will depend on the PC’s specifications. In this laboratory tutorial, the source compilation method will be used.

"WHAT IS INSIDE THIS BOX IS WHAT IS WRITTEN IN THE TERMINAL WINDOW"

THE TERMINAL WINDOW CAN BE OPENED BY USING (CTRL+ALT+" T")

Where to find Ubuntu installation and how to install the Robot Operating System (ROS) Indigo:

Software Steps Preparations

Step 1: Install Ubuntu

Current required version: Ubuntu 14.04

Ubuntu 14.04 for ROS Indigo (which is Recommended) and Ubuntu 12.04 for ROS Hydro / Groovy

Follow the standard Ubuntu Installation Instructions for 14.04 (Desktop):

- Download the Installer Image file, by picking the "Desktop CD" image appropriate for your machine:
  - 64-bit (AMD64): [http://releases.ubuntu.com/trusty/ubuntu-14.04.5-desktop-amd64.iso](http://releases.ubuntu.com/trusty/ubuntu-14.04.5-desktop-amd64.iso)
▪ Create an Ubuntu Live USB installer by burning the installer image onto a USB stick.
  ▪ Windows: http://www.ubuntu.com/download/desktop/create-a-usb-stick-on-windows
  ▪ Mac OS X: http://www.ubuntu.com/download/desktop/create-a-usb-stick-on-mac-osx
  ▪ Ubuntu: http://www.ubuntu.com/download/desktop/create-a-usb-stick-on-ubuntu
▪ Follow the Ubuntu Installation Instructions here:
  here: https://help.ubuntu.com/community/GraphicalInstall

Step 2: Install ROS

Current recommended version: ROS Indigo

ROS Indigo (Recommended)ROS HydroROS Groovy (RSDK 1.0 Compatible)

Install ROS Indigo

Configure Ubuntu repositories:

Configure your Ubuntu repositories to allow "restricted", "universe" and "multiverse." Follow the Ubuntu instructions guide for configuration.

Likely, they are already configured properly, and you only need to confirm the configuration.

Setup your sources list

$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu trusty main" > /etc/apt/sources.list.d/ros-latest.list'

Setup your keys

$ wget http://packages.ros.org/ros.key -O - | sudo apt-key add -

Verify Latest Debians

$ sudo apt-get update

Install ROS Indigo Desktop Full
$ sudo apt-get install ros-indigo-desktop-full

**NOTE:** You may get a prompt about 'hddtemp' during the installation. You can safely answer 'No'.

**Initialize rosdep**

rosdep enables you to easily install system dependencies for the source you want to compile and is required to run some core components in ROS.

$ sudo rosdep init
$ rosddep update

**Install rosinstall**

$ sudo apt-get install python-rosinstall

**Step 3: Create Baxter Development Workspace**

**Create ROS Workspace**

$ mkdir -p ~/ros_ws/src

# ros_ws (short for ROS Workspace)

**Source ROS and Build**

ROS Indigo (Recommended) ROS Hydro ROS Groovy

**Source ROS Setup**

$ source /opt/ros/indigo/setup.bash

**Build and Install**

$ cd ~/ros_ws
$ catkin_make
$ catkin_make install

**Step 4: Install Baxter SDK Dependencies**
(This step is not required if you are setting up your workstation on Baxter over SSH)

ROS Indigo (Recommended) ROS Hydro ROS Groovy

**Install SDK Dependencies**

```
$ sudo apt-get update  
$ sudo apt-get install git-core python-argparse python-wstool  
    python-vcs tools python-rosdep ros-indigo-control-msgs ros-indigo-joystick-drivers
```

**Step 5: Install Baxter Research Robot SDK**

**Current recommended installation: 1.2.0 Source**

1.2.0 Workstation Source (Recommended) 1.2.0 On-Robot Source 1.1.1 Indigo Debian 1.1.1 Hydro Debian 1.0 (Groovy) Workstation Source 1.0 Groovy Debian.

Install Baxter SDK

The `wstool` workspace tool is used to check all required Baxter Github Repositories into the workspace source directory.

```
$ cd ~/ros_ws/src

$ wstool init .
$ wstool merge https://raw.githubusercontent.com/RethinkRobotics/baxter/master/baxter_sdk.rosinstall
$ wstool update
```

**Source ROS Setup**

```
'''You must use ROS Indigo to use RSDK 1.2.0. Use command below.'''

# ROS Indigo  
$ source /opt/ros/indigo/setup.bash
```

**Build and Install**

```
$ cd ~/ros_ws  
$ catkin_make  
$ catkin_make install
```
Step 6: Configure Baxter Communication/ROS Workspace

This step describes the configuration and setup of your ROS environment. This section assumes an already configured Network Setup.

The current recommended environment setup: Baxter.sh ROS Environment Setup

Baxter.sh ROS Environment Setup (Recommended)

The baxter.sh script is a convenient script which allows for intuitive modification of the core ROS environment components. This user edited script will allow for the quickest and easiest ROS setup.

Download the baxter.sh script

```
$ wget https://github.com/RethinkRobotics/baxter/raw/master/baxter.sh
$ chmod u+x baxter.sh
```

Customize the baxter.sh script

Then edit the baxter.sh shell script making the necessary modifications to describe your development PC.

Using your favorite editor (gedit used for example)

```
$ cd ~/ros_ws
$ gedit baxter.sh
```

Edit the 'baxter_hostname' field

Baxter's hostname is defaulted as the robot's serial number. The serial number can be found on the back of the robot, next to the power button.

Alternatively, you can find your robot's hostname by plugging a USB keyboard into the back of Baxter and pressing Ctrl+Alt+F3.

```
# Specify Baxter's hostname
**baxter_hostname="14642.student.hig.se"**
```

Edit the 'your_ip' field

Modify where 'your_ip' is the IP address of your PC.
**your_ip="130.243.XXX.XXX"**

In our robot in the Lab: The ip is =130.243.14.27 (This is a fixed ip address for Baxter robot in the university’s network.

**Useful command for identifying your IP address:**

```
$ ifconfig
# Result will be contained in the 'inet addr' field (Ubuntu) or 'inet' field (Gentoo Robot PC).
```

Alternatively, you may choose to use the hostname of your development PC rather than the PC's IP address. For instructions, press Expand on the right.

**Verify 'ros_version' field**

Verify that the the “ros_version” field matches the ROS version you are running:

This field will default to "indigo"

***ros_version="indigo"***

**Save and Close baxter.sh script**

Please save and close the baxter.sh script.

**Initialize your SDK environment**

From this point forward, your ROS environment setup should be as simple as sourcing the baxter.sh script from the root of your Catkin workspace:

```
$ cd ~/ros_ws
$ . baxter.sh
```

**Step 7: Verify Environment**

A useful command for viewing and validating your ROS environment setup is:

```
$ env | grep ROS
```

The important fields at this point:

**ROS_MASTER_URI** - This should now contain the robot's hostname.

In our case for the Baxter robot is the hostname (14642.student.hig.se)

**ROS_IP** - This should contain your workstation's IP address.
or

**ROS_HOSTNAME** - If not using the workstation's IP address, the ROS_HOSTNAME field should contain the PC's hostname. Otherwise, this field should not be available.

**ROS setup**

**Description**

You've set up your Baxter and Development PC. These steps will show you how to communicate with Baxter, moving the arms, and allowing him to say "Hello!" to your lab.

**Step 1: Setup ROS Environment**

Upon compilation of our catkin workspace, the generated *devel* folder in the root of our catkin workspace will contain a setup script which informs the ROS environment of our compiled catkin workspace and the packages therein. The root of the catkin workspace that we created is located at `~/ros_ws`.

```
# Move to root of our catkin workspace
$ cd ~/ros_ws
```

If the *devel* folder is not available in the root of your *catkin_ws*, your catkin/ROS workspace (`~/ros_ws`) is not yet compiled:

ROS IndigoROS HydroROS Groovy

```
$ source /opt/ros/indigo/setup.bash
catkin_make
```

**Source ROS Environment Setup Script**

_Baxter.sh_ ROS Environment Setup

```
# Source baxter.sh script
$ . baxter.sh
```

**Step 2: Verify ROS Connectivity**

In this step, we will verify communication to and from Baxter and our development workstation.

**Verify ROS Master Ping**

The development workstation must be able to resolve the ROS Master (running on Baxter). This is defined by the `ROS_MASTER_URI` which has been set. This
ROS_MASTER_URI is typically the ROBOT_HOSTNAME configured (your robot serial number by default).

```bash
# Identify the ROS Master location
$ env | grep ROS_MASTER_URI
```

The result
(example): ROS_MASTER_URI=http://011303P0017.local:11311 011303P0017 .local ,in this case, is the location of our ROS Master. The development PC must be able to resolve and communicate with this Master.

Verify ability ping our ROS Master:

```bash
$ ping <our ROS Master>
# Example
$ ping 011303P0017.local
```

**View Available ROS Topics**

Now that we have identified our ROS Master is available on our network, let us view the available rostopics:

```bash
$ rostopic list
```

**Verify Development Workstation Ping**

We have verified communication from the development pc to Baxter. Now we will SSH to Baxter and verify communication from Baxter to the development PC.

Baxter must be able to resolve ROS_IP or ROS_HOSTNAME (only one should be set) which has been set.

Identify your ROS_IP or ROS_HOSTNAME set (ROS_IP was previously recommended):

**ROS_IP**

```bash
# Identify your ROS_IP
$ env | grep ROS_IP
```

The result (example): ROS_IP=130.243.14.27

130.243.14.27 in this case it is the IP address which must be resolvable to Baxter and all other ROS processes.

**Echo a ROS Topic**
Now that we have seen the available rostopics, and have verified communication to and from Baxter and our development PC, we can connect to and echo the contents being published to a chosen topic as in the following way:

Double check that our workspace is still configured correctly:

Baxter.sh ROS Environment Setup
Standard bashrc ROS Environment Setup

$ cd ~/ros_ws
$ . baxter.sh

**Echo Baxter's joint_states**

$ rostopic echo /robot/joint_states

You should see a continuous stream of Baxter's joint names with measured positions, velocities and torques.

All is well! You have successfully setup communication between your development PC and Baxter.

**Step 3: Enable the Robot**

The enable_robot tool is a fundamental tool used when working with Baxter which is provided in the baxter_tools SDK package, allows for enabling/disabling/resetting/stoping the robot. Baxter must be enabled in order to actively command any of the motors.

Enable the robot:

$ rosrun baxter_tools enable_robot.py -e

Baxter will now be enabled. The joints will be powered, and Baxter will hold its current joint positions within a position control loop.

Grabbing Baxter's cuff that can be seen in Figure 1:

*FIGURE 1: BAXTER ROBOT CUFF BUTTON [1]*
Enters Baxter's arms into "Zero-G" mode. The position control loop will be released with solely gravity compensation enabled. This allows for intuitive hand-over-hand guidance of the limbs throughout the workspace.

**Step 4: Run an Example Program**

A number of Baxter example programs are provided which use the `baxter_interface` package that contains Python modules for Baxter Research Robot development.

One of them can be tested by running this command:

```
$ rosrun baxter_examples joint_velocity_wobbler.py
```

This example will simply move the arms to a neutral position, enter into velocity control mode, moving each joint through a random sinusoidal motion.

After getting this movement and the robot's arms start to move so you are successful, and it is the time for the second laboratory tutorial.

**Checkpoints:**

1. What is the other alternative if you want to work with ROS and you only have your own PC with windows operating system?
2. What is the SDK and how can you enter FSM and what is that?
3. Is it practical to work on the default workspace in ROS and if not? Why?
4. When do we need to enable Baxter robot and how?
5. How can we finish running a Baxter example?

This laboratory tutorial is based on the:

**Reference:**


Laboratory tutorial 2: ROS concepts, workspace and packages

The targets of this laboratory tutorial are:

- Work with ROS and create ROS workspaces and Packages.
- Understanding ROS nodes.

1. ROS Workspace and ROS package:

After installing Ubuntu 14.04 and the ROS Indigo and before running some baxter_example in the ROS environment. It is useful to start with some basic knowledge about how to create the workspace and the package. The difference between these two concepts can be found in the thesis, see pages (16-17)

It is needed to create a catkin workspace since all ROS node and the Python codes should be within the packages that would be created. These are the most used steps by ROS users.

First, we start the laboratory tutorial by creating a directory for our new workspace. let the workspace be: ros_workspace/laboratory tutorial 2 in the home directory.

This ros_workspace workspace will contain several laboratory tutorials and for this laboratory tutorial 2, it will contain a folder laboratory tutorial 2 that contains all the nodes and the codes for the laboratory tutorial. Each time you want to have a package or a new project, you create your own package with a new name under the ros_workspace. Let us say: laboratory tutorial 2, laboratory tutorial 3, picknplace or sorting_batteries etc.

As we know from the theoretical part in this thesis that a package contains three folders: build, devel and src. We create a source folder src inside laboratory tutorial 2 then:

Open the new shell terminal and we go to the folder src

$ cd ros_workspace/laboratory tutorial 2/src
$ catkin_init_workspace

This will make the workspace ros_workspace/laboratory tutorial 2 as the initial workspace and as a result, it will create a file called CMakeLists.txt that describes how to build the code and where to install it to.

Now, it is the time to add missing two files with source file src. These files are build and devel. We can add them automatically by compiling the whole package and by going to laboratory tutorial 2 and run:
$ catkin_make

ROS uses these files to store information that is important to build the packages in *build* and the binary executables files in *devel*.

After the previous steps, you will be able to create your own packages in the same *src*.

You can create your package (let it be *project1*) in the *src*. First, we need to go to *src* and from that we run:

$ catkin_create_pkg project1 std_msgs rospy roscpp geometry_msgs turtlesim

By examining *package.xml* you will see the dependencies created and the *build* binary information.

You can try creating your own package (let it be: *lab2*) and don’t add any type of the dependencies and repeat the same procedure. Examine *package.xml* now,

**Checkpoint:**

1. Do you see any changes? What are these and why?

Now, in order to make the package ready to use with the ROS nodes and files so it needs the last step to build it. This can be done by running *catkin_make*. This command needs to be run in *laboratory tutorial 2*

$ catkin_make

It would take a few seconds and it would print some *build* information and configuration for the new package *laboratory tutorial 2*.

If everything is done correctly, so you will find *devel* directory contains the executable file “*setup.bash*” and to source *src*. This file will prepare the ROS package to be the used in this workspace *laboratory tutorial 2*.

To do that, run the following command

```
echo $ROS_PACKAGE_PATH
source devel/setup.bash
```

**Checkpoint:**

1. What is the difference if you run the first line again? why?
as a result of your steps that you have done, we will see the following structure for your files in the home directory, see Figure 2:

```
ros_worshipace
  |--- laboratory tutorial 2
  |     |--- build
  |     |--- devel
  |     |     |--- setup.bash
  |     |--- src
  |     |     |--- CMakeList.txt
  |     |--- Lab1
  |     |     |--- CMakeList.txt
  |     |     |--- Package.xml
  |     |--- Project1
  |     |     |--- CMakeList.txt
  |     |     |--- Package.xml
  |     |     |--- Include
  |     |     |--- src
```

**FIGURE 2: ROS WORKSPACE ARCHITECTURE**

### 2: Understanding ROS nodes:

We have mentioned some words like: *nodes, package* and now is the time for new words as [1]:

- **Nodes**: A node is an executable file within ROS package or the project in ROS.
- **Messages**: The heart of the communication in subscribing or publishing to a topic.
- **Topics**: Nodes can publish messages to a topic as well as subscribe to a topic to receive messages.
- **Master**: Name service for ROS that has the information about all nodes with ROS and through the master node can see each other.

In order to see the nodes that are running in the ROS, we run this command:

```
$ rosnode list
```
The error message will be displayed:

**ERROR:** Unable to communicate with master!

It means that the ROS master is not active. In order to activate it and to understand the ROS master, we run this command in a new terminal window:

```
$ roscore
```

New lines will be displayed that show the core service which has been started[/rosout]

We leave this terminal open and we open a new shell terminal (CTRL+ALT+T) and we run the code again to see the activated nodes.

```
$ rosnodes list
```

We will find one node which is activate after running the ROS master. This node is /rosout

To get more information about this node and what it has for duty inside ROS, we can run the command:

```
$ rosnodes info /rosout
```

The result for this command will be:

<table>
<thead>
<tr>
<th>Node [/rosout] : The node name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publications:</td>
</tr>
<tr>
<td>* /rosout_agg [rosgraph_msgs/Log]</td>
</tr>
<tr>
<td>Subscriptions:</td>
</tr>
<tr>
<td>* /rosout [unknown type]</td>
</tr>
<tr>
<td>Services:</td>
</tr>
<tr>
<td>* /rosout/get_loggers</td>
</tr>
<tr>
<td>* /rosout/set_logger_level</td>
</tr>
</tbody>
</table>

That shows the node /rosout publishes the /rosout_agg topic and subscribe messages from the topic /rosout.

**NOTE:**

The topic and the node, in this case, have the same name /rosout.
By using these commands, we get an idea about the ROS master and the ROS node and how to get the information about a specific node.

Now we can move to something which can be more interested in the ROS learner. We are trying to run a new node that is called *turtlesim*

**Turtlesim**

Run the command in a new shell terminal:

```
$ rosrun turtlesim turtle_node
```

You will get the result as shown in Figure 3.

By opening a new shell terminal and repeating the same commands that is illustrated previously and run the command `rosnodet list`, we will get a new node which is called “/turtlesim”

The same thing to see the node information via running the command:

```
$ rosnodet info /turtlesim
```

It will show new information about the publisher and the subscriber that is used in this node.

**Checkpoints:**

1. What is the difference between the previous node /rosout? What is the turtle1/cmd_vel? What is cmd_vel mean?

2. What is the connection between these nodes and the nodes that are used in Baxter robot? Any idea?

3. Do these nodes any relationship with the nodes used with Baxter robot?

In the above example, you couldn’t do so much with your robot “turtlesim”

Now you can run the turtlesim and try to control it using the keyboard. The idea behind this one can be found in pages (15-16) talking about the communication in ROS and the robot.

Use the command:
$ rosrun turtlesim turtle_teleop_key

Teleop: means the Tele operation that gives the \texttt{cmd\_vel} to the turtlesim which makes it move around.

The same example in controlling Baxter robot directly via the keyboard will be illustrated later which has the same idea as this example.

The result after running this code and drive the turtle around is shown in Figure 4:

**Checkpoints:**

1. What are the values that the turtle gets?
   when we press the arrow in the keyboard?
2. Command the turtle to move around?
3. What do the plus and minus values mean?

\textit{FIGURE 4: TURTLESIM WINDOW AFTER MOVING THE TURTLESIM AROUND}

\textbf{3. Understanding rqt\_graph:}

\textit{rqt\_graph} is a good tool in ROS that visualizes the computation graph, and this can be done by:

$ rosrun rqt\_graph rqt\_graph

The result of this code will be illustrated as Figure 5 the \texttt{/teleop\_turtle} is publishing a message control which is called \texttt{/turtle1/cmd\_vel} to the turtlesim that the turtlesim is receiving this message and subscribe it.

\textit{FIGURE 5: ROS COMPUTATION GRAPH}
Using rostopic echo

One of the rostopic tool is the rostopic echo that is used to visualize the messages that are sent to the topic.

Use the command

```
$ rostopic echo /turtle1/cmd_vel
```

Now in order to visualize the message that is sent to the topic /turtle1/cmd_vel

We move the turtle by using the keyboard and observe the message that is published to the topic. Then we use the rqt_graph window and we click the refresh button.

This refresh button will show the second node (rostopic node) which is subscribed to /turtle1/cmd_vel topic as shown in Figure 6

![ROS COMPUTATION GRAPH](image)

**FIGURE 6 : ROS COMPUTATION GRAPH**

**NOTE:**

We can see all the topics published or subscribed to by all node if we use the below command:

```
$ rostopic list
```
Checkpoints:

1. What are the node, topic and a message? What are they? Give some practical examples?

2. Can you try to run two turtles on the same screen?

   Repeat the steps illustrated and run the turtlesim and control the turtle around.

3. What is the most useful tool to see the message info?

Reference:

Laboratory tutorial 3: Baxter examples and training Baxter and FSM mode

The targets of this laboratory tutorial are:

• Run Baxter example codes that are written in python language and are uploaded in Rethink Robotics website.
• Learn the possibility of Baxter robot in practical examples and read the python code.
• Give the knowledge about Baxter environments and some definitions like: Source Development Kit (SDK) and the Field Service Menu (FSM).
• Training Baxter robot for a simple task and know how to enter the FSM menu.

Good to know

Before running any example, The Software Development Kit (SDK) should be installed and this installation has been explained in the first laboratory tutorial.

All Baxter examples are available in the link below:

https://github.com/RethinkRobotics/baxter_examples.git

Clone the SDK Examples

To clone the SDK examples, Rethink Robotics recommends by suggest cloning the 'baxter_examples' into your ROS development workspace (~/ros_ws/src). This will allow you to dig into the examples, follow along with the code walkthroughs and base your custom code off these examples.

$ cd ~/ros_ws/src
$ git clone https://github.com/RethinkRobotics/baxter_examples.git

• All SDK examples should be run after initializing the environment

Initialize

Initialize your environment:

$ cd ~/ros_ws
$ ./baxter.sh
Run a program

$ rosrun baxter_examples <example_program> [arguments]

Baxter robot must be enabled before beginning to run these examples:
To enable the Baxter robot, we run this command:

$ rosrun baxter_tools enable_robot.py -e

You will see that Baxter has a power to lift its arm to some position and the arms has resistance to any force if we will not enter Baxter's arms into "Zero-G" mode.

A number of Baxter example programs are provided which use the baxter_interface package which contains Python modules for Baxter Research Robot development.

Some Baxter examples:

1. Joint Velocity Wobbler:
   This example will simply move the arms to a neutral position, enter the velocity control mode, moving each joint through random sinusoidal motion. More about this example on the Joint Velocity Wobbler Example Page.
   
   $ rosrun baxter_examples joint_velocity_wobbler.py

2. Joint position keyboard
   Start the joint position keyboard example program, ex:
   
   $ rosrun baxter_examples joint_position_keyboard.py

Checkpoints:

1. Here you can test the singularity.
2. What is the singularity?
3. Puppet

$ rosrun baxter_examples joint_velocity_puppet.py -l right

Grab the other arm (e.g. the left in this case) at the cuff position and move it around using '0 G' mode to puppet the arm specified on the command line.

This example will show the capability of the Baxter robot in terms of the velocities in each joint. If the velocity is moved in one joint so it would replicate in the other joint with exact the same velocity. In order to change the amplification of the speed we can write the command:

$ rosrun baxter_examples joint_velocity_puppet.py -l right -a 2.5

And try the same example in the beginning and see the difference.

4. Keyboard Control

Start the example joint_position program using the keyboard to control Baxter:

$ rosrun baxter_examples joint_position_keyboard.py

5. Ik service example

In order to run the ik_service_client.py demo from the inverse kinematics package, with an argument of 'left' or 'right' arm, we can run this code:

$ rosrun baxter_examples ik_service_client.py -l left

The test will call the IK Service for a hard-coded Cartesian Pose for the given arm. The response will then be printed to the screen with the found joint solution and a True/False if the solution is a valid configuration for Baxter's arm.

This example gives an overview and introduction to the IK concept.

The following is an IK code, try to study it and run it in your Baxter robot.

```python
#!/usr/bin/env python
import rospy
from moveit_msgs.srv import GetPositionIK, GetPositionIKRequest, GetPositionIKResponse
```
from geometry_msgs.msg import PoseStamped
from moveit_commander import MoveGroupCommander
import numpy as np
from numpy import linalg

def main():
    # Wait for the IK service to become available
    rospy.wait_for_service('compute_ik')
    rospy.init_node('service_query')

    # Create the function used to call the service
    compute_ik = rospy.ServiceProxy('compute_ik', GetPositionIK)

    while not rospy.is_shutdown():
        raw_input('Press [ Enter ]: ')   

        # Construct the request
        request = GetPositionIKRequest()
        request.ik_request.group_name = "left_arm"
        request.ik_request.ik_link_name = "left_gripper"
        request.ik_request.attempts = 20
        request.ik_request.pose_stamped.header.frame_id = "base"

        # Set the desired orientation for the end effector HERE
        request.ik_request.pose_stamped.pose.position.x = 0.5
        request.ik_request.pose_stamped.pose.position.y = 0.5
        request.ik_request.pose_stamped.pose.position.z = 0.0
        request.ik_request.pose_stamped.pose.orientation.x = 0.0
        request.ik_request.pose_stamped.pose.orientation.y = 1.0
        request.ik_request.pose_stamped.pose.orientation.z = 0.0
request.ik_request.pose_stamped.pose.orientation.w = 0.0

try:
    #Send the request to the service
    response = compute_ik(request)

    #Print the response HERE
    print(response)

    group = MoveGroupCommander("left_arm")

    # Setting position and orientation target
    group.set_pose_target(request.ik_request.pose_stamped)

    # TRY THIS
    # Setting just the position without specifying the orientation
    ###group.set_position_target([0.5, 0.5, 0.0])

    # Plan IK and execute
    group.go()

except rospy.ServiceException, e:
    print "Service call failed: %s"%e

#Python's syntax for a main() method
if __name__ == '__main__':
    main()
Checkpoints:

1. You can give new values for x, y and z and see the difference. What are these values?
2. Can you explain the code communication according to publisher and subscriber concepts? What are the publisher and subscriber?

SDK Foundations (Interfaces)

The SDK provides interfaces that control the robot hardware layer via the Robot Operating System (ROS) to any other computers on the robot's network. By using the ROS network layer API, any client library or program that can "speak ROS" can control the robot directly. For a deeper understanding of the SDK System architecture see:

SDK System Overview

By using the foundation ROS interface layer, the robot can be controlled and programmed in any Programming Language that supports ROS. This has led to several user-created interface libraries in different languages. In addition, the Python Baxter Interface provides a Python Class-based interface library, which wraps many of the ROS interfaces in Python Classes.

Important Definitions:

ROS Interface

The ROS Interface is the foundation upon which all interaction with the Baxter Research Robot passes through and all other interface layers are built upon. The ROS layer is accessible over the network via any of the standard ROS Client Libraries such as rospy (Python) or roscpp (C++).

ROS API Reference

The ROS API describes the base interface layer for direct access from the command line or any language.

Python Interface

The Baxter SDK also offers a Python API via the baxter_interface module. This module wraps the ROS Interfaces in component-based Python Classes. The SDK Examples are written using this library in order to demonstrate how to use the robot interfaces.

Baxter Interface Overview

Gives a basic introduction to each class and component in the Python interface library 'baxter_interface'.
Code API

View the generated Python Code API docs for the baxter_interface module.

The Field Service Menu (FSM)

The Field Service Menu (FSM) is a pre-boot configuration menu that allows the user to do advanced tasks such as check network interface configuration, change the robot computer's hostname and run low-level hardware checks. The FSM menu can be seen as Figure 7:

```
Configuration

Hostname: baxter
Addr Type: DHCP
Address: <automatic>
Netmask: <automatic>
Gateway: <automatic>
DNS Server: <automatic>
MAC Address: 80:b1:1c:a3:5c:6f
ROS Name: ROS_HOSTNAME.local

Edit Hostname
Set Date, Time, Timezone, NTP Servers
Set ROS Naming Type
Set IP Address Type
Edit IP Address
Edit Netmask
Edit Gateway
Edit DNS Server
Delete stored task data

Save Changes
Revert Unsaved Changes

Return to Main Menu
```

**FIGURE 7: THE FIELD SERVICE MENU (FSM) ON THE ROBOT HEAD [3]**

Overview

The primary purpose of the Field Service Menu (FSM) is to give the user access to low-level, computer configuration options on the robot, especially:

**Edit Hostname**

Edit Network Configuration

**Set Static IP/DNS**

Set NTP Servers

**Set ROS naming convention**

Edit Timezone

**Change Run Mode (RSDK vs Demo Mode)**
The FSM also provides the ability to run low-level hardware tests to do light-weight verification of hardware components that sit below the robot application software level.

**QuickStart - FSM Activation / Deactivation**

**Accessing the FSM**

The FSM can only be accessed by hitting a specific key combination during robot boot-up:

1. Turn off Baxter with the Power button. Wait for complete shutdown (no lights, no noise).
2. Plug in a USB keyboard to Baxter's USB port (on rear).
3. You will need to hit `<Alt+FF>` on the keyboard during the boot-up to trigger the FSM:
   - Turn Baxter back on by hitting the Power button.
   - Once you see the Rethink Animation as it is shown in Figure 8...

![FIGURE 8: BEFORE ENTERING THE FIELD SERVICE MENU (FSM) ON THE ROBOT HEAD [3]](image)

On your keyboard, start repeatedly hitting the key combination `<Alt+FF>`

*(Hit and hold the Alt key, press down on the F key, release, press down on the F key again, release both and repeat).*

4. Repeat this key combination until you see the FSM screen comes up as shown in Figure 9:
If you hit a screen that says "Loading...", you missed your chance.

Use the Up/Down arrow keys to change menu selections, and the Enter key to go into a menu or select an option.

**Exiting: Disabling the FSM**

**Important:** Once the FSM is activated, it will come up every boot thereafter -- unless Disabled before exiting the menu.

To disable the FSM from reappearing next boot:

1. Select the 'Disable field service menu' option and hit Enter as it is in Figure 10.

You should now see a line near the top saying: On next boot FSM will be disabled

1. You can now leave the FSM through any of the normal Exit Actions, such as Reboot Robot.
- **Menu Functions**

After Activating the FSM, the top level of the Field Service Menu provides access to three sub-menus and shutdown/reboot options.

- **Configuration Menu**

This menu provides access to Baxter's hostname, time zone, and network configuration information. The Information here is useful for setting up Baxter on your Network, see Figure 11.

![Configuration Menu](image)

**FIGURE 11: HOW TO CHANGE THE HOSTNAME [3].**

- **Hostname** - Rename your Baxter (i.e. change robot computer's hostname)

  Changing the Hostname will also change Baxter's network identity name.

  Upon reboot, Baxter will then be accessible via the new name followed by '.local', rather than by the serial number (default hostname). In the case shown in the screenshot, Baxter's new avahi address would be test.local.

- **Addr Type** - Change the network addressing method from DHCP to Custom

  Toggling this will set Address, Netmask, Gateway, and DNS Server to blank, and enable editing them

- **ROS Naming** - Change how the master publishes node locations

Options are:
- ROS-Hostname
- ROS-Hostname.local
- ROS-IP

See the Networking page for more information on network configurations.

*Tip*: Make sure to Save Changes after editing any Configuration options, and then Shutdown or Reboot Baxter from the FSM main menu to apply the changes.

- **Date, Time, Timezone, NTP Servers**, see Figure 12

  ![FIGURE 12: TO EDIT THE DATE, TIME AND TIME ZONE [3].](image)

  **Time zone** - Set the robot's clock to your local time zone. This will primarily affect the time listed in the log files.

  - **NTP Servers** - Set your pool of NTP servers.

  - **Applications Menu** see Figure 13.

  The Applications menu launches the RSDK software application from the FSM.

  ![FIGURE 13: THE APPLICATION MENU [3]](image)

  - The 'Robot GUI' action will launch the Xdisplay on Baxter and display the Baxter Research Robot logo.
The 'RSDK Demo Mode' action will launch the Demo Mode

**Tests Menu**

The Tests menu provides a number of low-level hardware tests that can be run to verify the embedded hardware throughout Baxter. See Figure 14

*PC file permission check (at next reboot)*
CPU_health_check
Check JCB configuration
Repair JCB configuration
JCB_Firmware_version
JCB_sensor_errors
TBs
Graf_buttons
Gripperfinders
I-stop
Floor_mat
Foot Switch
Main LEDs
Camera_configuration
Camera_functional_test
Test left hand camera only
Test right hand camera only
Write_default_head_camera calming
Update_camera_firmware
Update_JCB_firmware
Restore JCB data blocks
Update_end_effector_firmware
Check_Gripper_Firmware_version
**Return to main menu**

**FIGURE 14: TO RUN THE TEST FOR DIFFERENT HARDWARE AND SOFTWARE PARTS OF THE ROBOT [3]**

- To run a test, select the test from the menu and follow the on-screen instructions.

- **Exit Actions**
  - **Disable field service menu**
  
  Select to disable the FSM from appearing at boot-up.

**NOTE:** By default, accessing the FSM will enable it for future boots.

- **Reboot Robot**

  Shuts down the robot computer and hardware, then restarts the robot, re-flashing the controller boards and configurations if needed.

- **Shutdown Robot**

  Shuts down the entire robot. Use to power off the robot or do a hard restart of the robot to ensure new configurations or firmware are written to the hardware.

**NOTE:** You can also leave the FSM via the Applications menu to go directly to normal
RSDK operation.

With the Demo version and not the programming in Python. The basic functions of holding, lifting and simple positioning tasks could be completed successfully.

Working with Baxter

Now after we have our robot all set up and we've installed the Baxter SDK on workstation and setup ROS workspace environment, we want to know more about working with the robot and using the workspace and tools to develop our own programs and research.

Why training the robot?

One of the strongest features of the Baxter robot comparing with the other industrial robot is that the Baxter robot can be programmed by any co-worker after a few minutes of training. In other words, there is no need for programmers to write the code that controls the robot.

By this simple feature, the Baxter robot becomes more efficient in both the production and saving the time as well as the money if any maintenance needed. This feature is limited in the FSM that is used in this work.

How to train the Baxter robot

One of the most beneficial features of the Baxter robot is the ability to learn and repeat the motions by demonstration these steps of actions to the robot. This procedure doesn’t need a pre-programming or to have the skills in programming.

Even a layperson can train the Baxter robot. This training procedure is based on advanced artificial intelligence.

When a worker or the user moves the arm of the Baxter robot, the task will be stored in the robot’s memory which will be used later to repeat the behaviour.

How to Interact with Baxter

Using the Training Cuffs as shown in Figure 15.

Using the training cuffs to move the arms, to manipulate the state of the grippers, and secondarily, to select on-screen options.
Training cuff switch: Squeeze this switch at the indentation in the cuff to move the robot’s arm. When this switch is squeezed, the blue indicator on the arm’s navigator button lights up.

Grasp button: Press to toggle a parallel gripper open or closed, or a vacuum gripper on or off.

Action button: Press to select items on the display screen. Create waypoints, hold actions; select, copy, or move actions on the task map; create a new subtask; add/create landmarks; outline a visual search area.

Navigating the Screens

Use the navigator that is shown in Figure 16 on either of the arms to scroll to and interact with options on the screen. When you press the OK button (2) (or the action button on the cuff), the white indicators on the navigator light up.
Back button: Press to exit the current screen and return to the previous screen. Will also cancel the last action.

Knob: Scroll the knob to move between on-screen options. Press the knob (OK) to select an option.

OK indicator light: When the action button on the cuff or the OK button on the navigator is pressed, the white indicator around the knob lights up.

Rethink button: Press to display options for the current screen.

Training cuff indicator: When the switch on the cuff is squeezed, the blue indicators along the top and bottom edge of the navigator light up.

**Moving the Arms**

To move an arm, squeeze the cuff at the indentation just above the other buttons, and push or pull the arm to the location you want. Figure 17 shows where to squeeze the cuff of the arm.

Squeezing the cuff releases the tension and resistance in the arm, making it easier to manipulate. With its seven degrees of freedom an incredible amount of flexibility Baxter enhances arm stability by attempting to fix its elbow in position whenever the lower arm is moved.

**NOTE:** When the switch is pressed, the blue indicator lights illuminate on the corresponding navigators on the arm and torso.
When grasping the training cuff, you can move the arms by either repositioning the lower arm or changing the height of the elbow. Figure 18 show how to do it.

**Figure 18: Moving the Arm and Holding the Elbow [3]**

To move the lower arm: While squeezing the cuff (1), move the robot’s arm to the desired location.

To move the elbow: By design, the elbow (2) will try to maintain its current height and will spring back if you do not actively reset it. While squeezing the cuff, move the elbow to the desired position. Continue to hold the elbow at the new location and release the cuff. This will reset the elbow at the new position.

1. **Grasping Objects**

   Training involves showing Baxter how to pick up and place objects.

   To grasp an object: Position the gripper over the object, press Grasp.

   To release an object: With an object in the robot’s hand, press Grasp.

   To open or close the gripper without creating a pick or place: Without an object in hand, press Grasp twice quickly.
The following is a link on www.youtube.com that shows how to train the Baxter robot:

https://www.youtube.com/watch?v=4pdU2rCv91Q

Another way to train the robot arms but not the gripper is by running the following Baxter example:

We can notice the changes between the following training method and the previous one. To train Baxter robot, we run this command:

2. The joint position waypoints

Another way to train the Baxter’s arms is by running the joint position waypoints example:

Verify that the robot is enabled from an RSDK terminal session, ex:

$ rosrun baxter_tools enable_robot.py -e

Start the joint position waypoints example program, specifying left or right arm on which we will run the example:

$ rosrun baxter_examples joint_position_waypoints.py -l right

Moving the arm in zero-g mode, when we reach a joint position configuration we would like to record, press that limb’s navigator wheel.

You will then be prompted with instructions for recording joint position waypoints.

Press Navigator 'OK/Wheel' button to record a new joint joint position waypoint.
Press Navigator 'Rethink' button when finished recording waypoints to begin playback.
Moving the arm in zero-g mode, when we reach a joint position configuration we would like to record, press that limb's navigator wheel.

You will get the feedback that the joint position waypoint has been recorded:

Waypoint Recorded

When done recording waypoints, pressing the limb navigator's 'Rethink' button will begin playback.

[INFO] [WallTime: 1399571721.540300] Waypoint Playback Started
Press Ctrl-C to stop...

The program will begin looping through the recorded joint positions. When the previous joint position command is fully achieved (within the accuracy threshold), the next recorded joint position will be commanded.

Waypoint playback loop #1
Waypoint playback loop #2
...

Pressing Control-C at any time will stop playback and exit the example.

Arguments

Important Arguments:

- `l` or `--limb` : The limb (arm) on which we will be running the example.
- `s` or `--speed` : The joint position motion speed ratio [0.0-1.0] (default: = 0.3)
- `a` or `--accuracy` : The threshold in Radians at which the current joint position command is considered successful before sending the next following joint position command. (default: = 0.008726646)

See the joint position waypoint's available arguments on the command line by passing `joint_position_waypoints.py` the `-h`, help argument:

$ rosrun baxter_examples joint_position_waypoints.py -h
usage: joint_position_waypoints.py [-h] [-l {left,right}] [-s SPEED]
optional arguments:
- h, --help           show this help message and exit
- s SPEED, --speed SPEED
                        joint position motion speed ratio [0.0-1.0] (default:=0.3)
- a ACCURACY, --accuracy ACCURACY
                        joint position accuracy (rad) at which waypoints must achieve

required arguments:
- l {left,right}, --limb {left,right}
                        limb to record/playback waypoints

3. Trajectory playback example

The third way to train the robot’s arms is by running the robot joint trajectory interface, parse a file created using the joint position recorder example, and send the resulting joint trajectory to the action server.

Verify that the robot is enabled from an RSDK terminal session, ex:

```
$ rosrun baxter_tools enable_robot.py -e
```

Record a joint position file using the joint_recorder.py example, ex:

```
$ rosrun baxter_examples joint_recorder.py -f <example_file>
```

The recorder is now recording joint positions with corresponding timestamps for both arms. Move the arms while holding the cuffs.
NOTE: You can open and close the grippers while recording by using Baxter's cuff buttons: Oval = Close, Circle = Open

Press any key to exit when done recording.

Start the joint trajectory controller, ex:

```
$ rosrun baxter_interface joint_trajectory_action_server.py --mode velocity
```

In another RSDK terminal session, Run the joint trajectory playback example program, ex:

```
$ rosrun baxter_examples joint_trajectory_file_playback.py -f <example_file>
```

Both arms will then be commanded to repeat the trajectory recorded during the joint position recording. The difference between this example and the joint_position playback example is that the trajectory controller has the ability to honor the velocities (due to the timestamps) to more accurately repeating the recorded trajectory.

4. **Change an image to Baxter’s head**

The following command shows how to display an image (e.g. .png or .jpg) to Baxter’s head display. The Baxter display resolution is 1024 x 600 pixels.

```
$ rosrun baxter_examples xdisplay_image.py --file=`rospack find baxter_examples`/share/images/happyface.png
```

Replace the --file argument with the path to your own image and in the case of this thesis:

```
baxter_examples`/share/images/happyface.png
```

To see help for the xdisplay_image program, run:

```
$ rosrun baxter_examples xdisplay_image.py -h
```
5. Running a Python Code

Two methods are introduced here to run any Python code:

1. By saving the Python code in a folder somewhere in the computer and by running the Python code from the folder that contains this Python script.

2. By making the Python code, an executable code which can be done by running the following command:

   ```
   $ chmod a+x Pythoncode_name.py
   ```

In order to run a Python code that executes some tasks such as holding, lifting and positioning an object which is discussed in the thesis, it can be done by different procedures:

1. Saving the Python program somewhere in the computer and in the case of this work, it is saved in : ./Thesis/baxter_ws/src/picknplace/Baxter_works.py

   Then to execute the code Baxter_works.py, it can be done by running the following commands:

   ```
   $ cd Thesis/baxter_ws/src/picknplace
   $ python Baxter_works.py
   ```

   If the code does not have any error, it shows a few lines and some arguments will appear in the computer screen. We will see the head of the Baxter robot turns to the left and the right and a message that is telling us “The robot is enabled” appears before coming to the argument of either the Manual(m) or Automatic(a) running will be achieved.

2. There is another way to execute any Python code in ROS. This way needs to make the Python code “Baxter_works.py” in this work, to be executable by giving it the executable permission. This command is:

   ```
   $ chmod a+x Baxter_works.py
   ```

   When this code is running without any error, it means that “Baxter_works.py” is an executable code. We can then run “Baxter_works.py” either with the command:

   ```
   $ ./Baxter_works.py
   ```

   Or by using the command:
$ rosrun picknplace Baxter_works.py

Reference:

Appendix B: Python Code

Python code for holding, lifting and positioning 12 cubes that are arranged in an array (4*3) dimension and rearrange them into another array (3*4) dimension by using Baxter robot. The code is based on using the (7-dof) joints’ angles:

#!/usr/bin/env python
#The above tells Linux that this file is a Python script,
#and that the ROS should use the Python interpreter in
#/usr/bin/env
#to run it. using "chmod +x [filename]" to make
#this script executable is very important if the rosrun wants to
#be used under ROS.

# University of Gavle, Sweden
# EE470D | Master Thesis in Electronics/Automation
# Development of Python Algorithms to execute holding, lifting and Positioning task by Baxter Robot
#
# Student: Rabe Andersson
#
# 2019
#
# This is a program of is used to move objects that is arranged in two arrays (3*4) to (4*3) by using Baxter robot that shows the picking, holding and placing objects in different locations
#
#
# Import the rospackage. For an import to work, it must be specified in both the package manifest AND the Python file in which it is used.

import rospy
import baxter_interface

from baxter_interface import CHECK_VERSION

#Run this program as a new node in the ROS computation graph called /Pick_arm to execute Picking, holding and positioning the object

rospy.init_node("Pick_arm")
print("Getting robot state... ")
rs = baxter_interface.RobotEnable(CHECK_VERSION)
#To enable the robot in the beginning of the python script and before achieving the task
init_state = rs.state().enabled
print("Enabling robot before starting the action... ")
rs.enable()

h = baxter_interface.Head()

r = baxter_interface.Limb("right")
rightgripper = baxter_interface.Gripper('right')
P = {};PUp = {};T = {};TUp ={}
# All the positions that Baxter robot will move to and from in both building the leaving and picking the cubes from the certain places

P[1] = {'right_s0': 0.10431069357620813, 'right_s1':0.6059224112147384, 'right_w0': -2.196276993054941, 'right_w1': 1.291228328202547, 'right_w2': 0.1169660350762628, 'right_e0': 1.7867041226895357, 'right_e1': 0.825651590793239}
PUp[1] = {'right_s0': 0.0011504855909140602, 'right_s1':0.46901462589596526, 'right_w0': -2.1445051414638083, 'right_w1': 1.64711870991963, 'right_w2': 0.2998932440315984, 'right_e0': 1.896767244220314, 'right_e1': 1.168126369747426}
P[2] = {'right_s0': -0.07938350577307016, 'right_s1':0.744364177321397, 'right_w0': -2.38610711555111, 'right_w1': 1.4342720366728618, 'right_w2': 0.2511893540162365, 'right_e0': 1.7537235357499992, 'right_e1': 1.160839612322868}
PUp[2] = {'right_s0': -0.10699515995500761, 'right_s1':0.5752427954570302, 'right_w0': -2.1847721371458007, 'right_w1': 1.776732475682804, 'right_w2': 0.3328738309711347, 'right_e0': 1.912874042493111, 'right_e1': 1.3694613483847031}
P[3] = {'right_s0': -0.3344078117590202, 'right_s1':0.7370777685789413, 'right_w0': -2.3669323557071933, 'right_w1': 1.4143302864303515, 'right_w2': 0.3877136441380383, 'right_e0': 1.5010002009458774, 'right_e1': 1.4607332052638853}
PUp[3] = {'right_s0': -0.28532042654668693, 'right_s1':0.6247136758663347, 'right_w0': -2.1824711659639724, 'right_w1': 1.8150827672654157, 'right_w2': 0.41647578391088985, 'right_e0': 1.7928400458410774, 'right_e1': 1.6497963373707625}
P[4] = {'right_s0': -0.4931748233051605, 'right_s1':0.7110000951848893, 'right_w0': -2.318611960888803, 'right_w1': 1.4304370847031482, 'right_w2': 0.44178646691099915, 'right_e0': 1.35833987672534, 'right_e1': 1.6164322552342547}
PUp[4] = {'right_s0': -0.44677190447162674, 'right_s1':0.6408204741391316, 'right_w0': -2.173650776436315, 'right_w1': 1.8342575271139834, 'right_w2':
0.5100486119719001, 'right_e0': 1.6643691548556738, 'right_e1': 1.8699225804323194}

P[5] = {'right_s0': 0.0337475773334791, 'right_s1': 0.5948010505025692, 'right_w0': -2.177869223603163, 'right_w1': 1.2501943421266122, 'right_w2': -0.006902913545484362, 'right_e0': 1.804344901750218, 'right_e1': 0.7719758315033345}
PUp[5] = {'right_s0': 0.02454369260616662, 'right_s1': 0.5414952181235511, 'right_w0': -2.3128595329342327, 'right_w1': 1.7077041121134369, 'right_w2': 0.01457281748491143, 'right_e0': 2.118043972872785, 'right_e1': 1.028534118277171}

P[6] = {'right_s0': -0.2281796421979553, 'right_s1': 0.6872233929726653, 'right_w0': -2.263388652524928, 'right_w1': 1.3192234775814558, 'right_w2': 0.13192234775814557, 'right_e0': 1.674340029976929, 'right_e1': 1.1558545236716593}
PUp[6] = {'right_s0': -0.15569904997036949, 'right_s1': 0.666131517192409, 'right_w0': -2.3151605041160606, 'right_w1': 1.8016604353714185, 'right_w2': 0.0908836168221076, 'right_e0': 2.0716410540392514, 'right_e1': 1.334179790263385}

P[7] = {'right_s0': -0.3324903357741634, 'right_s1': 0.791534086548735, 'right_w0': -2.4037478946164324, 'right_w1': 1.4856603936070231, 'right_w2': 0.19673303604630432, 'right_e0': 1.6674371164314448, 'right_e1': 1.3219079436902552}
PUp[7] = {'right_s0': -0.3903981105168378, 'right_s1': 0.6273981422451342, 'right_w0': -2.20164592581254, 'right_w1': 1.77021382929197676, 'right_w2': 0.3551165523954733, 'right_e0': 1.806645872932046, 'right_e1': 1.6375244910676792}

P[8] = {'right_s0': -0.5595194923812047, 'right_s1': 0.6845389265938658, 'right_w0': -2.2921507922977793, 'right_w1': 1.405126401703039, 'right_w2': 0.31676703268933795, 'right_e0': 1.3778982427180728, 'right_e1': 1.5619759372643225}
PUp[8] = {'right_s0': -0.5196359918961839, 'right_s1': 0.5925000793207411, 'right_w0': -2.13645174232741, 'right_w1': 1.808946844113874, 'right_w2': 0.40420393760780654, 'right_e0': 1.6846944002951556, 'right_e1': 1.7916895602501632}

P[9] = {'right_s0': 0.05368932757598948, 'right_s1': 0.47169909227476475, 'right_w0': -1.9209274416295095, 'right_w1': 1.17196321944456, 'right_w2': -0.1971153124327566, 'right_e0': 1.6505633277647052, 'right_e1': 0.5503156076538922}
PUp[9] = {'right_s0': -0.0674951546669582, 'right_s1': 0.41724277430483253, 'right_w0': -2.167514832820896, 'right_w1': 1.6260196351585385, 'right_w2': 0.01457281748491143, 'right_e0': 1.951235621902463, 'right_e1': 0.9759952762920945}
P[10] = {'right_s0': -0.1903400722813244, 'right_s1': 0.6239466854723921, 'right_w0': -2.190524565100371, 'right_w1': 1.3119370688390002, 'right_w2': -0.004218447166688787, 'right_e0': 1.7103885784922364, 'right_e1': 0.9744612955040291}
PUp[10] = {'right_s0': -0.26537867630417655, 'right_s1': 0.4183932598957466, 'right_w0': -2.033291534342116, 'right_w1': 1.6517138133556193, 'right_w2': 0.15186409800065595, 'right_e0': 1.8093303393108455, 'right_e1': 1.240223467005357}
P[11] = {'right_s0': -0.45022336124436896, 'right_s1': 0.6377525125633607, 'right_w0': -2.1886070891155143, 'right_w1': 1.3092526024602007, 'right_w2': -0.004218447166688787, 'right_e0': 1.7103885784922364, 'right_e1': 0.9744612955040291}
PUp[11] = {'right_s0': -0.47821851062327775, 'right_s1': 0.41647578391088985, 'right_w0': -1.9960925002358947, 'right_w1': 1.6517138133556193, 'right_w2': 0.15186409800065595, 'right_e0': 1.8093303393108455, 'right_e1': 1.240223467005357}
P[12] = {'right_s0': -0.5882816321540562, 'right_s1': 0.68568941218478, 'right_w0': -2.2952187538735505, 'right_w1': 1.4200827143849217, 'right_w2': 0.13767477571271589, 'right_e0': 1.4511458253396015, 'right_e1': 1.463801166839656}
PUp[12] = {'right_s0': -0.45520879880499654, 'right_s1': 0.4943253089607457, 'right_w0': -2.0210196880390328, 'right_w1': 1.1922865673839378, 'right_w2': 0.7485826244880819, 'right_e0': 1.48719437385401489, 'right_e1': 1.026616642292313}
T[1] = {'right_s0': 0.6258641614572488, 'right_s1': 0.5445631796993219, 'right_w0': -2.081228433963535, 'right_w1': 1.2252671543349217, 'right_w2': 0.13767477571271589, 'right_e0': 1.4511458253396015, 'right_e1': 0.870150601787646}
TUp[1] = {'right_s0': 0.5250049246537829, 'right_s1': 0.2577087723647495, 'right_w0': -1.843849070382674, 'right_w1': 1.4894953450367368, 'right_w2': 0.3021942152134265, 'right_e0': 1.5374322446581559, 'right_e1': 1.7575584877197128}
T[2] = {'right_s0': 0.45520879880499654, 'right_s1': 0.4943253089607457, 'right_w0': -2.0210196880390328, 'right_w1': 1.1922865673839378, 'right_w2': 0.7485826244880819, 'right_e0': 1.48719437385401489, 'right_e1': 1.026616642292313}
TUp[2] = {'right_s0': 0.426446659032145, 'right_s1': 0.21015536794030168, 'right_w0': -1.7518060597651426, 'right_w1': 1.46917009959957255, 'right_w2': 0.7600874903972225, 'right_e0': 1.5428011774157548, 'right_e1': 1.190369091399081}
\[ T[3] = \{ 'right_s0': 0.3209547986502285, 'right_s1': 0.61896124170349875, 'right_w0': -2.18438641948829, 'right_w1': 1.2866366626277062, 'right_w2': 0.62584614572488, 'right_e0': 1.6336895390979655, 'right_e1': 1.084524170349875 \} 
TUp[3] = \{ 'right_s0': 0.273432075440575, 'right_s1': 0.3531990764106165, 'right_w0': -1.926679865840797, 'right_w1': 1.6010924473554007, 'right_w2': 0.670346043059258, 'right_e0': 1.6954322658103536, 'right_e1': 1.313087554429914 \} 

\[ T[4] = \{ 'right_s0': 0.1867621609250491, 'right_s1': 0.7136845615636888, 'right_w0': -2.29826715449321, 'right_w1': 1.3391652278239663, 'right_w2': 0.510321075440575, 'right_e0': 1.6954322658103536, 'right_e1': 1.313087554429914 \} 
TUp[4] = \{ 'right_s0': 0.1487961364248851, 'right_s1': 0.3716068458652414, 'right_w0': -1.9351167639174494, 'right_w1': 1.6068448753099709, 'right_w2': 0.6622962051695274, 'right_e0': 1.6954322658103536, 'right_e1': 1.3571895020816198 \} 

\[ T[5] = \{ 'right_s0': 0.5100486119719001, 'right_s1': 0.6139581035313462, 'right_w0': -2.169432329266946, 'right_w1': 1.3391652278239663, 'right_w2': 0.510432107168714, 'right_e0': 1.718825472825606, 'right_e1': 1.150485599094603 \} 
TUp[5] = \{ 'right_s0': 0.431815591789744, 'right_s1': 0.2527233348041219, 'right_w0': -1.8714565612202048, 'right_w1': 1.540883701430898, 'right_w2': 1.0921943209744145, 'right_e0': 1.5734807931734631, 'right_e1': 1.4741555371578825 \} 

\[ T[6] = \{ 'right_s0': 0.3474466484560462, 'right_s1': 0.4912573473203037, 'right_w0': -2.0678061020695377, 'right_w1': 1.2237331735355887, 'right_w2': 0.9326603190343316, 'right_e0': 1.3729128051574453, 'right_e1': 1.2927623089904325 \} 
TUp[6] = \{ 'right_s0': 0.3926990816986659, 'right_s1': 0.431815591789744, 'right_w0': -1.9995439570086369, 'right_w1': 1.2659176452024377, 'right_w2': 0.8551942892461182, 'right_e0': 1.419315723990979, 'right_e1': 1.193053577778805 \} 

\[ T[7] = \{ 'right_s0': 0.17985924737956477, 'right_s1': 0.6749515466698582, 'right_w0': -2.2407624359036182, 'right_w1': 1.3568060068846484, 'right_w2': 0.7420632061395689, 'right_e0': 1.5366652546246123, 'right_e1': 1.330293046724246 \} 
TUp[7] = \{ 'right_s0': 0.17717478100076528, 'right_s1': 0.32903887900142126, 'right_w0': -1.85880121972015, 'right_w1': 1.3568060068846484, 'right_w2': 0.7420632061395689, 'right_e0': 1.5366652546246123, 'right_e1': 1.330293046724246 \}
iter = 0

# To ensure that the Right gripper will be open before starting the action
rightgripper.open()
rospy.sleep(0.5)

# Moving the head to the right and left which imitating the human behavior in examining the environment.
h.set_pan(-1, speed=0.20, timeout=10)  # moving the head to the right with the speed of 20% of the full allowed velocity
h.set_pan(1, speed=0.20, timeout=10)  # moving the head to the left with the speed of 20% of the full allowed velocity

# To let the robot seems to look ahead towards the cubes with the speed 0.20
h.set_pan(0, speed=0.20, timeout=10)

# Argument that gives the user to choose between two alternatives: (A) automatic and (M) manual:
slc = raw_input('Do you want to run (a) automatic or (m) manual Choose either a or m ')

# The loop while to do the first phase of the program (Moving the cubes from the array (4*3) to the other location and building another array (3*4))
# Keep publishing until a Ctrl-C is not pressed
while not rospy.is_shutdown():
    
    # move from Ps(array(4*3)) to Ts(array(3*4))
    # if the manual (m) is chosen so enter this manual phase
    if slc == 'm':
        raw_input("Ready to make a move the cubes and the magazine is loaded, if so... Press Enter to continue...")
        print("Performing the cube moving action, Any time Ctrl+C quits the program")

    # This is a for loop for 12 cubes
    for i in range(1, 13):
        
        # To Move the head towards the object that the robot will pick
        h.set_pan(-0.25, speed=0.20, timeout=10)

        # Moving the arm to the position which is above the object
        # go to Pup
        r.move_to_joint_positions(PUp[i])
        print("PUp" + str(i) + "th UP location")
        rospy.sleep(0.5)

        # Moving the arm to down to the object
        # go to P
        r.move_to_joint_positions(P[i])
        print("P" + str(i) + "th location")
        rospy.sleep(0.5)
# Closing the gripper which means grasp the object if it is the gripper is used in the robot and suck the object and holding it if the vacuum cup gripper is used

# close gripper
rightgripper.close()
rospy.sleep(0.5)
print("gripper closed")

# Moving the arm up to the position which is above the object
# go to PUp
r.move_to_joint_positions(PUp[i])
rightgripper.close()
rospy.sleep(0.5)
print("PUp + str(i) + "th UP location")
rospy.sleep(0.5)

# To Move the head towards the new location that the robot will place the cubes
h.set_pan(0.25, speed=0.20, timeout=10)

# Moving the arm towards the TUp positions
r.move_to_joint_positions(TUp[i])
print("TUp + str(i) + "th location")
rospy.sleep(0.5)

# moving the arm towards the T position which means the new locations that the cubes will be places
# go to T
r.move_to_joint_positions(T[i])
print("T + str(i) + "th location")
rospy.sleep(0.5)

# open gripper
rightgripper.open()
print("gripper opened and Nr. of cubes that are left to the new place or array(3*4):" + str(i))
rospy.sleep(0.5)

#Moving to the TUp positions in order to move vertically
r.move_to_joint_positions(TUp[i])
print("TUp + str(i) + "th location")
rospy.sleep(0.5)

# Here is the second phase of the program to leave back the cubes from (3*4) array to the origin place that the program has started with
# move from Ts to Ps

# This is in case of choosing the (m)manual sequence
if slc == 'm':
    # Giving the ability to the user of either continue the second phase of
raw_input("Ready to leaving back the cubes to the start origin place? ...if Yes ...Press Enter to continue...")
print("Start moving back the cubes to the first start positions, Any time Ctrl+C quits the program")
# This is a for loop which starts from the last position 12 and ending with Position 1. Last object to the first object LIFO Last cube is in the array will be the first cube which will be moved to its original place.
for i in range(12, 0, -1):
    # To Move the head towards the position that the robot will place the cubes
    h.set_pan(0.25, speed=0.20, timeout=10)
    # Moving towards TUp positons
    r.move_to_joint_positions(TUp[i])
    print("TUp" + str(i) + "th location")
    rospy.sleep(0.5)
    # go to T
    r.move_to_joint_positions(T[i])
    print("T" + str(i) + "th location")
    rospy.sleep(0.5)

    # close gripper
    rightgripper.close()
    rospy.sleep(0.5)
    print("gripper closed and Nr. of cubes that are new place or array(3*4):" + str(i))
    rospy.sleep(0.4)
    # To Move the head towards the object that the robot will pick
    h.set_pan(-0.25, speed=0.20, timeout=10)
    # Moving towards the TUp positions
    r.move_to_joint_positions(TUp[i])
    print("TUp" + str(i) + "th location")
    rospy.sleep(0.5)
    # go to PUp
    r.move_to_joint_positions(PUp[i])
    rightgripper.close()
    rospy.sleep(0.5)
    print("PUp" + str(i) + "th UP location")
    rospy.sleep(0.1)

    # go to P
    r.move_to_joint_positions(P[i])
    print("P" + str(i) + "th location")
    rospy.sleep(0.1)

    # open gripper
    rightgripper.open()
    print("gripper opened")
    rospy.sleep(0.5)

    # go to PUp
    r.move_to_joint_positions(PUp[i])
    print("PUp" + str(i) + "th UP location")
    rospy.sleep(0.5)
    print("Finished the moving the cubes from (4*3) array to (3*4) array and vice versa :) !")

A49
Appendix C: Hardware specifications

More Description of Baxter robot

Baxter is a humanoid, anthropomorphic robot sporting two seven degree-of-freedom arms and state-of-the-art sensing technologies, including force, position, and torque sensing and control at every joint, cameras in support of computer vision applications, integrated user input and output elements such as a head-mounted display, buttons, knobs and more.

The Baxter Robot is designed for continuous operation and can run 24/7 for extended periods of time without the risk of damaging the hardware of the robot. It is recommended that the robot be connected to an uninterruptible power when running continuously in order to prevent hard drive corruption in the event of a power outage that can cause an improper shutdown of the robot. If the hard drive of the robot becomes corrupt due to an improper shut down, a fresh install (or factory reset on robots running v1.2 or later) will be required.

Maximum Joint Speeds

<table>
<thead>
<tr>
<th>Joint</th>
<th>Maximum Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>2.0</td>
</tr>
<tr>
<td>S1</td>
<td>2.0</td>
</tr>
<tr>
<td>E0</td>
<td>2.0</td>
</tr>
<tr>
<td>E1</td>
<td>2.0</td>
</tr>
<tr>
<td>W0</td>
<td>4.0</td>
</tr>
<tr>
<td>W1</td>
<td>4.0</td>
</tr>
<tr>
<td>W2</td>
<td>4.0</td>
</tr>
</tbody>
</table>

*TABLE 1: MAXIMUM JOINT SPEEDS (RAD/SEC)*

Joint Flexure Stiffness

<table>
<thead>
<tr>
<th>Joint</th>
<th>Stiffness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small Flexures (W0, W1, W2)</td>
<td>3.4deg @ 15Nm (~250Nm/rad)</td>
</tr>
<tr>
<td>Large Flexures (S0, S1, E0, E1)</td>
<td>3.4deg @ 50Nm (~843Nm/rad)</td>
</tr>
</tbody>
</table>

*TABLE 2: FLEXURE STIFFNESS (K)*
S1 Spring Specifications

<table>
<thead>
<tr>
<th>Description</th>
<th>Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spring Type</td>
<td>JIS standard die spring: ASF 35 X 200</td>
</tr>
<tr>
<td>Free Length</td>
<td>200 mm</td>
</tr>
<tr>
<td>Stiffness (K)</td>
<td>9.6 N/mm</td>
</tr>
<tr>
<td>Operating length</td>
<td>101 mm - 154 mm</td>
</tr>
</tbody>
</table>

**TABLE 3: S1 SPRING SPECS**

Joint Sensor Resolution

The resolution for the joint sensors is 14 bits (over 360 degrees); so \(360/(2^{14})\) = 0.021972656 degrees per tick resolution.

All of the joints have a sinusoidal non-linearity, giving a typical accuracy on the order of +/- 0.10 degrees, worst case +/- 0.25 degrees accuracy when approaching joint limits. In addition, there may be an absolute zero-offset of up to +/- 0.10 degree when the arm is not calibrated properly. Be sure to tare and calibrate the arms if you're trying to minimize accuracy errors in the joint sensors.

**NOTE:** The performance of the joint controllers is a separate matter; ultimately, the accuracy of the controller is only limited by the accuracy of the sensors. In a case where joint position controllers are included; we use a threshold to determine when the joint states are "close enough" to the commanded joint angles to call it acceptable. In the baxter_interface for the RSDK, we use a default threshold of: JOINT_ANGLE_TOLERANCE = 0.00872664626 radians (0.5 degrees), which is set in settings.py and used by the limb interface and the joint_position examples. If you want to improve the accuracy, you can write your own controller to adjust the setpoint and to overcome any steady-state error in the internal low-level controllers. Even when the joint controller is slightly off the target position, it always knows exactly how far off it is, via the joint position sensors.

**Peak Torque** The peak torque specification refers to the maximum amount of torque that can be applied to each joint.

<table>
<thead>
<tr>
<th>Joint</th>
<th>Peak Torque</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0,S1,E0,E1</td>
<td>50Nm</td>
</tr>
<tr>
<td>W0,W1,W2</td>
<td>15Nm</td>
</tr>
</tbody>
</table>

**TABLE 4: PEAK TORQUE PER JOINT**

A51
Other Hardware Specifications

Camera Specifications

<table>
<thead>
<tr>
<th>Description</th>
<th>Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Resolution</td>
<td>1280 x 800 pixels</td>
</tr>
<tr>
<td>Effective Resolution</td>
<td>640 x 400 pixels</td>
</tr>
<tr>
<td>Frame Rate</td>
<td>30 frames per second</td>
</tr>
<tr>
<td>Focal Length</td>
<td>1.2mm</td>
</tr>
</tbody>
</table>

* TABLE 5: CAMERA SPECIFICATIONS *

On Board CPU

<table>
<thead>
<tr>
<th>Description</th>
<th>Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>3rd Gen Intel Core i7-3770 Processor (8MB, 3.4GHz) w/HD4000 Graphics</td>
</tr>
<tr>
<td>Memory</td>
<td>4GB, NON-ECC, 1600MHZ DDR3</td>
</tr>
<tr>
<td>Hard Drive</td>
<td>128GB Solid State Drive</td>
</tr>
</tbody>
</table>

* TABLE 6: ON BOARD CPU *

Component Weights

<table>
<thead>
<tr>
<th>Description (with pedestal)</th>
<th>Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total weight</td>
<td>298 lbs / 135.2 kg</td>
</tr>
<tr>
<td>One Arm</td>
<td>47 lbs / 21.3 kg</td>
</tr>
<tr>
<td>Torso</td>
<td>70 lbs / 31.8 kg</td>
</tr>
<tr>
<td>Pedestal</td>
<td>134 lbs / 60.8 kg</td>
</tr>
</tbody>
</table>

* TABLE 7: COMPONENT WEIGHTS *

Electrical Power

<table>
<thead>
<tr>
<th>Description</th>
<th>Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Battery Operation</td>
<td>DC-to-120V AC Inverter (Note: the Baxter robot has an internal PC, which cannot be powered directly off of 24V DC)</td>
</tr>
<tr>
<td>Interface</td>
<td>Standard 120VAC power. Robot power bus and internal PC both have &quot;universal&quot; power supplies and support 90 - 264V AC (47 - 63Hz)</td>
</tr>
<tr>
<td>Max Consumption</td>
<td>6A at 120V AC, 720W max per unit</td>
</tr>
<tr>
<td>Electrical Efficiency</td>
<td>87% to 92%</td>
</tr>
</tbody>
</table>

A52
Power Supply | Uses medical-grade DC switching power supply for robot power bus
---|---
Tolerance to sags | Sags tolerated to 90V. Sustained interruption will require manual power-up
Voltage Flicker | Holdup time 20mS
Voltage Unbalance | Single phase operation only

**TABLE 8: ELECTRICAL POWER**

**Miscellaneous Specifications**

<table>
<thead>
<tr>
<th>Description</th>
<th>Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Screen Resolution</td>
<td>1024 x 600 pixels</td>
</tr>
<tr>
<td>Positional Accuracy</td>
<td>+/- 5 mm</td>
</tr>
<tr>
<td>Max Payload (including end-effector)</td>
<td>5 lb / 2.2 kg</td>
</tr>
<tr>
<td>Gripping Force (max)</td>
<td>35N or 8 lbs</td>
</tr>
<tr>
<td>Infrared Sensor Range</td>
<td>1.5 – 15 in / 4 – 40 cm</td>
</tr>
</tbody>
</table>

**TABLE 9: MISCELLANEOUS SPECIFICATIONS**

This Appendix is based on: