Universidade de Aveiro Departamento de Engenharia Mecânica
2017

Jorge Manuel das Neves Martins de Sousa

Estimação de Verticalidade Usando a Estabilização de uma Cabeça Humanóide

Verticality Estimation Using Stabilization of a Humanoid Head

Universidade de Aveiro Departamento de Engenharia Mecânica
2017

Jorge Manuel das Neves Martins de Sousa

Estimação de Verticalidade Usando a Estabilização de uma Cabeça Humanóide

Verticality Estimation Using Stabilization of a Humanoid Head

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Mecânica, realizada sob orientação científica de Vítor Manuel Ferreira dos Santos, Professor Associado do Departamento de Engenharia Mecânica da Universidade de Aveiro e de Filipe Miguel Teixeira Pereira da Silva, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

O júri / The jury

Presidente / President

Prof. Doutor Rui António da Silva Moreira
Professor Auxiliar da Universidade de Aveiro

Vogais / Committee

Prof. Doutor João Paulo Morais Ferreira
Professor Adjunto do Instituto Superior de Engenharia de Coimbra

Prof. Doutor Vítor Manuel Ferreira dos Santos
Professor Associado da Universidade de Aveiro (orientador)

Agradecimentos / Acknowledgements

Em primeiro lugar quero agradecer aos meus orientadores, o professor Vítor Santos e o professor Filipe Silva, pelo apoio que me prestaram durante esta dissertação, tanto no sentido académico como no sentido afetivo. A todos os professores que tive durante o curso, pois foram um contributo essencial para a minha formação pessoal e académica. Ao Eng. Festas, pelo tempo dispendido na maquinação das peças desenhadas e na correção dos erros nos desenhos mecânicos.

À minha família, que sempre me apoiou durante todo o meu percurso académico, acreditando que podia alcançar tudo o que almejasse. Aos amigos que fiz durante esta passagem pela Universidade de Aveiro, especialmente os colegas do Laboratório de Automação e Robótica .

Por fim, gostaria também de agradecer à Motofil por ceder o manipulador robótico da FANUC que foi de extrema importância para as experiências feitas neste trabalho.

Palavras-chave

Estimação de Verticalidade; Estabilização da Cabeça; Robótica Humanóide; Controlo de Realimentação; Rastreamento Visual.

Resumo

O Projeto Humanóide da Universidade de Aveiro (PHUA) é a base desta dissertação que tem como objetivo estudar a estabilização da cabeça de um robô humanóide para efeitos de equilíbrio. Foram assim desenvolvidas soluções integradas usando o Robot Operating System (ROS). Para estabilizar a cabeça humanóide, foi estimado o vetor da gravidade (verticalidade) usando dados visuais. Para o fazer, duas soluções distintas foram abordadas, ambas baseadas na biblioteca Visual Servoing Platform (ViSP). Depois de estimada a verticalidade, a cabeça humanóide (uma câmara montada numa unidade roll-tilt) foi estabilizada recorrendo a um controlador proporcional-derivativo (PD) de posição para o servomotor responsável pelo ângulo roll e a um controlo PD de velocidade para o servomotor responsável pelo ângulo tilt. Para fazer as experiências, o sistema foi montado num manipulador robótico para permitir repetibilidade e controlo preciso de movimentos. A análise dos resultados das experiências mostra que quer a estimação da verticalidade quer a estabilização da cabeça humanóide foram tarefas realizadas com sucesso. Esta análise permite também concluir que a taxa de aquisição de imagem influencia os resultados, e que a limitada taxa do sistema usado (cerca de 15 imagens por segundo) condicionou um pouco a robustez dos resultados em situações mais exigentes.

Keywords Verticality Estimation; Head Stabilization; Humanoid Robotics; Feedback Control; Visual Tracking.

Abstract The Humanoid Project of the University of Aveiro (PHUA) is the basis of this dissertation, whose purpose is to study the effects of the stabilization of a humanoid head in obtaining its equilibrium. Therefore, integrated solutions using the Robot Operating System (ROS) were developed. To stabilize the humanoid head, the gravity vector (verticality) was estimated using visual data. In order to do so, two distinct solutions were approached, both based on the Visual Servoing Platform (ViSP) library. After estimating verticality, the humanoid head (a camera mounted on a roll-tilt unit) was stabilized using a proportional-derivative (PD) controller for the position of the servo responsible for the roll angle and a PD controller for the velocity of the servomotor responsible for the tilt angle. To perform the experiments, the system was mounted on a robotic manipulator to allow repeatability and precise movement control. An analysis to the experiment results shows that both the verticality estimation and head stabilization tasks were performed successfully. This analysis also allows for the conclusion that the image frame rate influences results, and that the limited frame rate of the system used (about 15 frames per second) slightly conditioned the robustness of results in more demanding scenarios.

Contents

List of Tables

List of Figures

List of code snippets

Acronyms

DoF Degrees of Freedom. 1, 14

IMU Inertial Measurement Unit. 2–4, 7, 11–14

LAR Automation and Robotics Laboratory. 1, 4, 10, 14, 20, 31, 34, 62

OpenCV Open Source Computer Vision Library. 17, 26–28

PD Proportional-Derivative. 24, 37, 47, 48, 51, 53, 56, 57, 59–62

PHUA Humanoid Project of the University of Aveiro. 1, 3, 12, 14, 61–63

PTU Pan-Tilt Unit. 3, 4, 10–12, 21, 23, 24

ROS Robot Operating System. 4, 16, 17, 24–29, 31–34, 36, 38, 39, 47, 61

RTU Roll-Tilt Unit. 23, 24, 34–37, 39, 42, 44, 45, 47, 48, 53, 56, 57, 60–62

TP Teach Pendant. 40, 47, 48, 57

ViSP Visual Servoing Platform. 4, 19, 29, 31, 62

Chapter 1

Introduction

"The development of the automobile replaced the horse as a mobility tool, and the development of humanoid robots offers the promise to replace humans as the bearer of hard and dull tasks." - Shuuji Kajita, Hirohisa Hirukawa, Kensuke Harada, Kazuhito Yokoi. [1]

1.1 Background/Motivation

In a world where robotics is developing at an extraordinary rate, be it on a more industrial or domestic environment, there are more and more reasons to explore and solve the problems that are caused by this development. Solving these problems can bring great benefits to the world's economy if, for example, these problems are about industrial robotics. However, if we focus on robots to be used in a domestic environment, their main purpose would be to improve our day-to-day activities and to do it while interacting with humans safely.

Humanoid robots (see Fig. 1.1) can be considered part of this domestic application, and have been a point of interest for many decades. Although new applications for this kind of robots are being found, interest in this topic began with the writing of sci-fi stories, where androids and the moral intricacies regarding their existence were the main plot. The first application of humanoid robotics was a military one, at the *DARPA Rescue Challenge* in 2013 [2], where multiple teams of engineers built robots with the purpose of entering highly hazardous environments and performing some kind of complex task. However, if the cost of building a human-like robot goes down in the near future, mass commercialization of humanoids does not seem like an impossibility at all.

At the University of Aveiro, research on humanoid robots is present under the Humanoid Project of the University of Aveiro (PHUA). This project comes from a partnership between the Department of Mechanical Engineering (DEM) and the Department of Electronics, Telecommunications and Informatics (DETI) and its purpose is to provide students who join the Automation and Robotics Laboratory (LAR) with a way to learn and practice in the areas of programming and robotics.

PHUA started in 2004 [4] and since then students have worked to find new and better solutions, ending up with the structure shown in Fig. 1.2. This structure weighs about 6 kg, has a height of 667 mm and has 27 Degrees of Freedom (DoF), which allow it to

Figure 1.1: Humanoid robots. Taken from [3].

perform a range of human-like motions. It also features a total of eight pressure sensors under its feet (four for each), two cameras on its head and an Inertial Measurement Unit (IMU), consisting of accelerometers, gyroscopes and magnetometers [5]. A haptic interface was also developed, allowing the robot to be controlled using force feedback [6]. Lately, students have tried to integrate visual and inertial data in order to assist humanoid balance [7], and visual servoing using fixed targets [8].

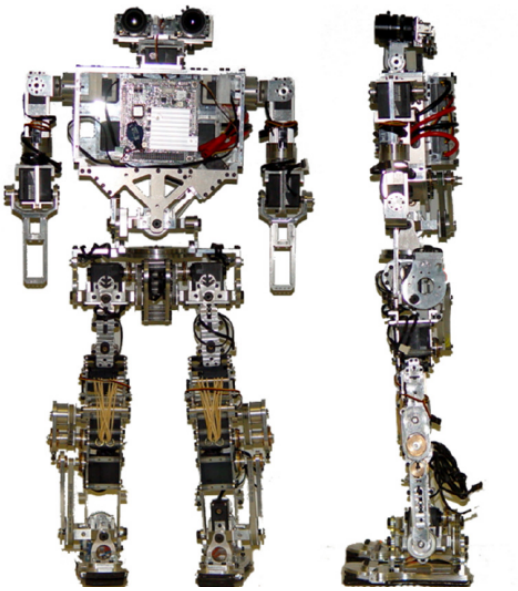


Figure 1.2: Last iteration of the PHUA project. Taken from [9].

1.2 Problem description

The main problem with humanoid robots is associated with the ability to maintain balance during both standing and walking. Since they have legs to emulate human

motion, there is a lack of a fixed reference point in relation to their environment (as opposed to other types of robots). Therefore it is difficult to determine the angular position and velocity of humanoid robot's joints at any given moment in time.

This problem has been approached in two ways by the PHUA project until now. The way which has received the most attention to date consists of finding the Center of Pressure (CoP) by using load cells under the robot's feet, thus allowing to know where the robot is falling towards (e.g., if more force is felt at the "toes" then the robot should be falling to the front) [10].

This approach, however, has not been fully successful and therefore attention has switched to another way of solving the problem, where the objective is to first stabilize the head of the humanoid, so that afterwards it is easier to balance the whole robot. In 2016, this topic was approached by another author [7] but more work needed to be done in order to understand if it is a viable way to approach the problem, or if there is yet another way to achieve the humanoid's balance.

1.3 Objectives

In order to achieve head stabilization, one must find the direction of the gravity vector (verticality). To do so, an existing Pan-Tilt Unit (PTU) with a camera and 2 IMU's installed [8] will be used to obtain visual and inertial data, as was done before [7]. However, less focus will be given to inertial data since there was a great interest in further exploring the benefits of visual servoing.

Regarding the visual data, it will be analyzed in a different way from what was done before, since the other approach based on visual features proved to be of a limited use.

In summary, the objectives proposed for this dissertation are the following:

- Estimate the direction of the gravity vector using both visual and inertial data.
- Develop computational tools which will act on the camera support (PTU), based on the previously estimated verticality, allowing it to remain stable.
- Explore and demonstrate the benefits of head stabilization when trying to achieve a humanoid robot's balance.

1.4 Related work

In this section, a short summary of the theses that support this work will be made and there will also be a brief explanation of how their results will be used.

1.4.1 Visual Servoing of a Humanoid Head, by *C. Sousa*

The first thesis that this work will be based on was written by César Sousa in 2016 [8] and its main subject is the visual control of a humanoid head. This thesis' main objectives are twofold: to develop image analysis techniques for detection and following of a static visual cue and to then try to control the camera so that even if exterior disturbances were provoked, it would still fixate on the target.

A PTU with a mounted camera was developed and built during the writing of that thesis and it will also be the one used for the current work. This PTU was meant to allow

the camera to track the target and its joints were controlled so that when an external disturbance was provoked (via a robotic manipulator) the camera would always point in the direction of the target.

It was also in this thesis that the first interaction with Visual Servoing Platform (ViSP) [11] was made, a modular C++ library which allows fast development of visual servoing applications. Since it was first released to public in 2005, this library has evolved immensely and its integration with the Robot Operating System (ROS) environment is now easier than ever. Therefore, this library will also be explored when trying to estimate verticality whilst using visual data.

1.4.2 Visual and Inertial Data Integration to Assist Humanoid Balance, by *J.Peixoto*

The second thesis that this work will be based on was also written in 2016 by João Peixoto [7] and its main subject is the integration of visual and inertial data to assist with humanoid balance. That thesis' main objective was to combine the visual data coming from the camera mounted in the previously mentioned PTU with the inertial data coming from two different IMU's, which were also installed on the PTU. The merged data would then allow for an estimation of the humanoid head's orientation, making it possible to stabilize and achieve the humanoid's balance in an easier way.

During that thesis' writing, the author arrived at the conclusion that the method he used to estimate verticality using local visual features couldn't provide great accuracy, although being well implemented. As such, an alternative to estimating verticality using visual data must be found during this work. Multiple experiments were also made with the IMU's installed on the PTU, allowing for the estimation of the gravity vector, although these measurements only provide accurate results when taken without external disturbances.

Additionally, a ROS node to communicate with the robotic manipulators at LAR was also developed, which will prove useful when conducting experiments to evaluate this work's solutions.

1.5 Balance in humans and humanoid robots

1.5.1 Human balance system

In order to understand the basics of balance and how to reach equilibrium on humanoid robots, it is paramount to understand how humans can balance themselves.

The sense of balance is a result of three major groups of sensory information being sent to the brain. These are sight, the vestibular system, and proprioception (body awareness). Of these, the vestibular system is the one of most importance and, therefore, will be explained in greater detail. The vestibular system is a region of the inner ear where three semicircular canals filled with a fluid called endolymph are located (see Fig. 1.3). These canals are all perpendicular to each other and, at the base of each canal, there is an enlarged portion called ampulla, which contains hair cells (the crista) that are affected by movement. These hair cells are bent (sending a nervous impulse) when our body starts or stops moving, and when it accelerates or decelerates, or when it changes

direction. These canals function just as a modern day accelerometer does, helping us to maintain equilibrium while moving. [12]

Also in the inner ear, there exist two membranous sacs called utricle and saccule (see Fig. 1.3). Within them exist the same kind of hair cells embedded in a gelatinous membrane with tiny crystals of calcium carbonate called otoliths. These otoliths are pulled by gravity and bend the hair cells, allowing them to send nervous signals to the cerebellum and giving us information about the orientation of our heads while standing still. Another analogy can be made here with gyroscopes, which give us the orientation of the gravity vector. [12]

In short, the otolith organs provide us information about the position of the body at rest, while the semicircular canals give us informations about the body in motion. [12]

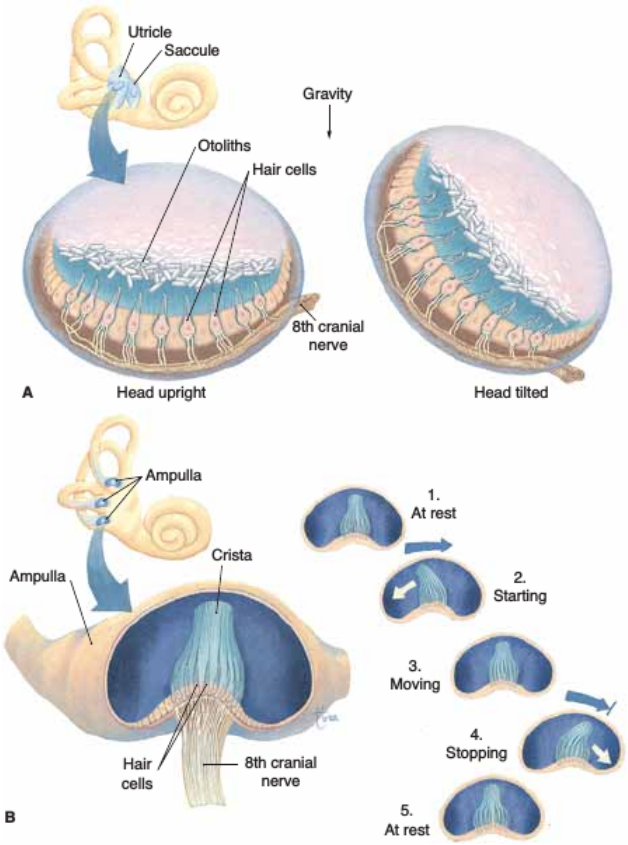


Figure 1.3: Physiology of equilibrium. (A) Utricle and saccule. (B) Semicircular canals. Taken from [12].

When it comes to sight influencing balance, it can be said that it is only an accessory to the vestibular system. However, since this work focuses mainly on estimating verticality (and therefore reaching equilibrium) through visual data, the vision system will also be explored.

In the eye, there are sensory receptors called rods and cones (see Fig. 1.4) which are located in the retina. Rods detect only the presence of light, whereas cones detect colors. Rods are also proportionally more abundant toward the periphery of the retina. This

is why our best vision in dim light or at night, where colors can't be seen, is periphery vision. When light strikes the rods and cones, they send impulses through the optic nerve to the brain that provide visual cues which identify how a person is oriented relative to the environment. [12][13]

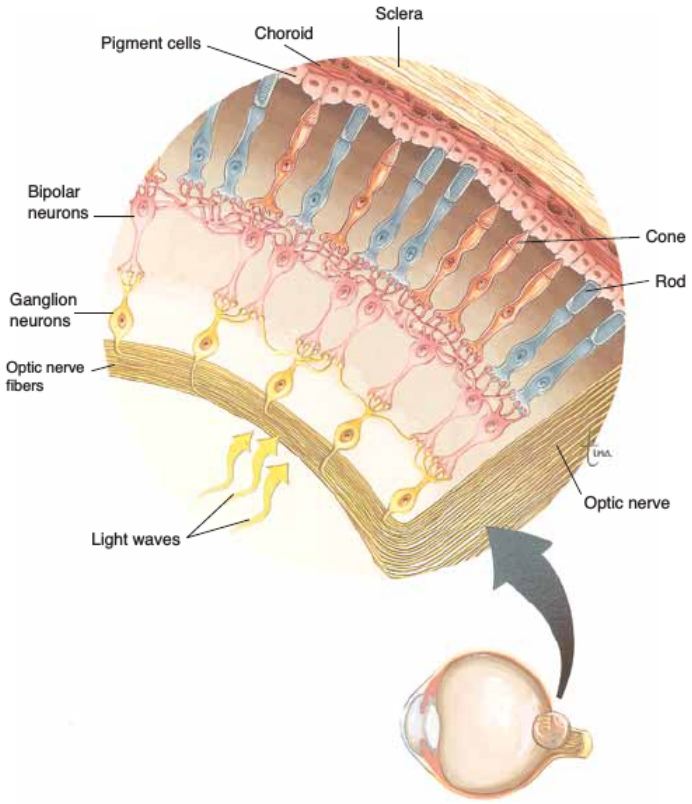


Figure 1.4: Microscopic structure of the retina. Taken from [12].

After information is gathered from the vestibular and sight systems (and also the proprioceptive system which is not going to be explained), it is sent to the cerebellum, which is responsible for the maintenance of posture and equilibrium through an involuntary process. The cerebellum then sends nervous signals to muscles all over the body which maintain balance and also to muscles controlling the eyes, in what is known as the Vestibulo-Ocular Reflex (VOR). This reflex helps stabilize gaze during active head movements (e.g., running) and passive head movements (e.g., sitting on an accelerating car). [13]

As an extra note, it is useful to know that disorientation occurs when the vestibular organs, the eyes and the muscles and joints send conflicting information to the cerebellum. One example of this is when a person is sitting on a car which is accelerating and deccelerating and reading a book at the same time. Since the eyes do not see any difference between consecutive images but the vestibular system senses the movement of the car, the cerebellum doesn't know what signals to send to the body, since it doesn't know if it is standing still or moving. [13]

1.5.2 Achieving balance in humanoid robots

Now that the basics of how humans perceive balance have been explained, this section will try to describe previous works on the topic of balancing humanoid robots.

The large majority of humanoid robots us the concept of Zero Moment Point (ZMP) as the base of their balance mechanisms. ZMP was first defined by Vukobratović and it is the point where the resultant of the forces inflicted on the robot's foot passes the surface of the foot (see Fig. 1.5) [14]. It is therefore a criterion to judge if the contact between the sole of the foot and the ground can be kept. The contact is kept if the ZMP is an internal point on the sole. [1]

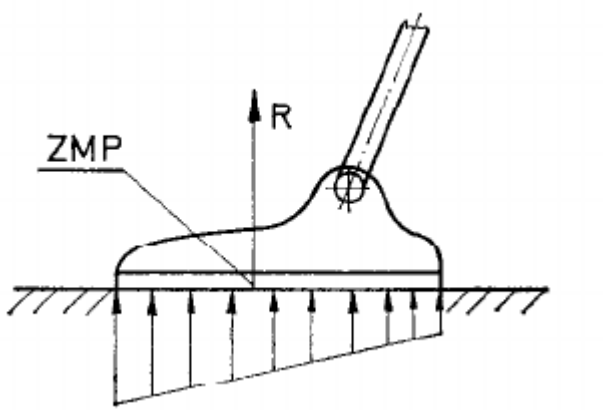


Figure 1.5: Zero Moment Point. Taken from [14].

If the robot is not moving, then the ZMP coincides with the projection of the Center of Mass (CoM) of the robot onto the ground. Most humanoid robots use the concept of ZMP to plan motion patterns that can make the robot walk while keeping the contact between the sole of the supporting foot and the ground [1].

However, as was said before, this dissertation will focus on an approach different from what has been done before, making it difficult to find similar applications implemented, be it in the University of Aveiro, or all around the world. After extensive research, only one paper [15] was found that also explored the benefits of head stabilization when trying to achieve a humanoid robot's balance. It states that stabilizing a humanoid head during locomotion brings benefits when estimating the gravitational vertical. In this paper, verticality was estimated using a damped inclinometer and an IMU and results were obtained through the use of simulations. As such, it is possible to conclude that there is room for further study in a real world application which could use other methods (e.g. using visual data) to estimate verticality.

Chapter 2

Experimental Infrastructure

In this chapter, the experimental infrastructure used throughout the dissertation will be explained in detail, including hardware and software specifications.

At the beginning of this dissertation, all of the hardware was connected to each other just as it was in [7] (Fig. 2.1). For a more detailed explanation of how the different components were integrated with each other please refer to chapter 2 (Experimental Tools and Setup) of that work. Some changes had to be made in order to fulfill this work's objectives and they are described in section 3.1.

The next two sections describe each of the components of the *final* solution.

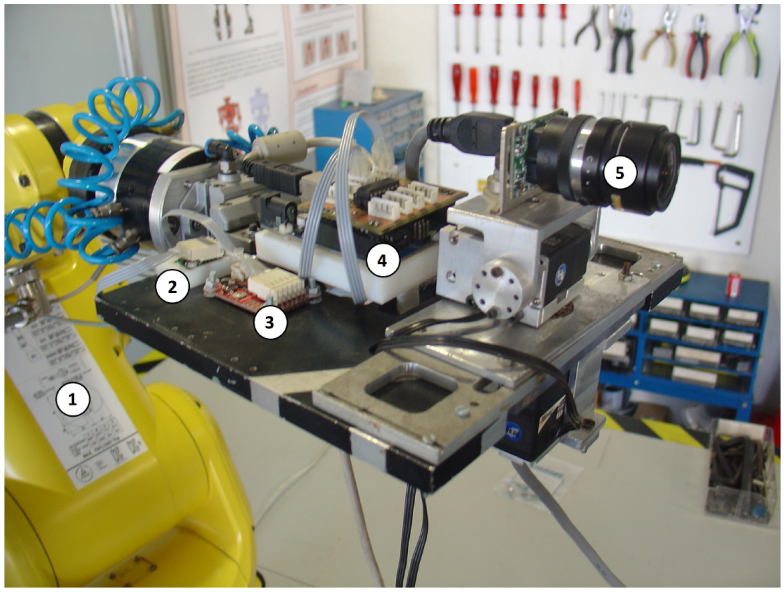


Figure 2.1: Detailed view of the initial experimental infrastructure.
1 - FANUC manipulator; 2 - POLOLU-minIMU 9DOF v2; 3 - RAZOR 9DOF - SEN 10736; 4 - Arduino UNO R3; 5 - Firefly MV-03MTC - Pointgrey. Taken from [7].

2.1 Hardware

2.1.1 PointGrey Flea3 2.8 MP Color GigE Vision (Sony ICX687)

The camera which was already mounted on the PTU was a FFMV-03M2C-CS model made by PointGrey. However, this camera is becoming obsolete, since the FireWire 1394a communication protocol is not supported by modern day laptops anymore. As such, it was decided a new camera had to be installed. The camera chosen was the Flea 3 GigE (short name), one that was already available in LAR, therefore reducing monetary and time costs. This camera is also made by PointGrey and is the model FL3-GE-28S4C-C (Fig. 2.2). The main specifications of this camera are shown in table 2.1.

Figure 2.2: The PointGrey Flea3 GigE camera. Taken from [16].

Table 2.1: PointGrey FL3-GE-28S4-C specifications. Taken from [16].

Specifications	**Values**
Max. Resolution	1928 x 1448
Frame rate	15 FPS
Chroma	Color
Sensor name	Sony ICX687
Sensor type	CCD
Readout method	Global shutter
Sensor format	1/1.8"
Pixel size	3.69 μm
Interface	GigE
Power requirement	12-24 V
Power consumption (maximum)	2.5 W
Dimensions	29x29x30 (mm)
Mass	38 grams (without optics)

2.1.2 HSR-5498SG Digital Robot Servo

The servomotors already present in the PTU were two Hitec HSR-5498SG Digital Robot Servo (Fig. 2.3). Their specifications were thought to be sufficient for this work's needs, so they were not replaced. Their specifications can be found in table 2.2.

Figure 2.3: An HSR-5498SG Digital Robot Servomotor. Taken from [17].

Table 2.2: HSR-5498SG Digital Robot Servo specifications. Taken from [17] and [18].

Specifications	**Values**	
Control system	Pulse width control (1500 μs neutral)	
Operating voltage range	6.0 to 7.4 V	
Operating temperature range	-20° to 60°C	
Test Voltage	6.0 V	7.0 V
Operating speed (No load)	0.22s/60°	0.19s/60°
Stall torque	11.0 kgf-cm	13.5 kgf-cm
Standing torque	10.6 kgf-cm/5° holdout	14.4 kgf-cm/5° holdout
Idle current	8 mA	10 mA
Running current (No load)	200 mA	240 mA
Stall current	1200 mA	1480 mA
Operating angle	0° to 180° (600 μs / 2400 μs respectively)	
Motor type	Cored Metal Brush / Nd magnet	
Dimensions	40 x 20 x 37 mm	
Weight	59.8 g	
Gear material	1 metal-karbonite & 3 steel	

2.1.3 Arduino UNO R3

To process the sensor readings coming from the IMU's, an Arduino UNO R3 board (Fig. 2.4) was used. Arduino is an open-source electronics platform, based on easy-to-use hardware and software. There are many different Arduino boards, but all of them are able to read inputs, make some decision based on those inputs, and then generate output signals accordingly [19]. These boards are based on a microcontroller that can be

programmed with the Arduino programming language [20] using the Arduino Software [21].

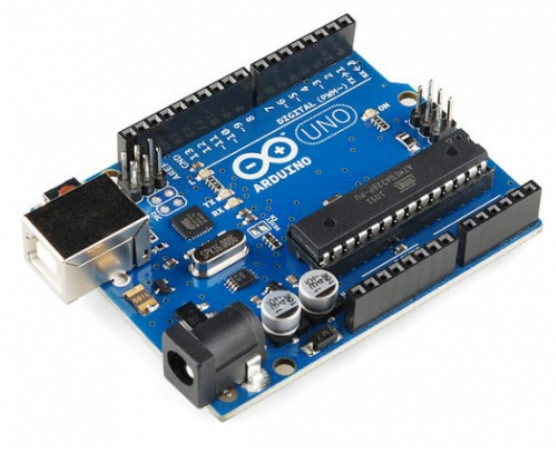


Figure 2.4: An Arduino UNO R3 board. Taken from [22].

The Arduino UNO R3 board comes with an ATmega328P microcontroller and it has the following features [22]:

- 14 digital input/output pins (of which 6 can be used as PWM outputs);
- 6 analog inputs;
- 16 MHz quartz crystal;
- USB connection;
- Power jack;
- ICSP header;
- Reset button.

2.1.4 Inertial Measurement Units

With the IMU's present in the PTU, it is possible to estimate its spatial orientation just by analyzing the inertial data that these sensors output. PHUA has a total of nine IMU's distributed along its body. However, only two of these were used in this work, since using any more would not be necessary.

A short introduction to each IMU present in the PTU is shown in the following pages.

RAZOR 9DOF - SEN 10736

The 9DOF Razor IMU (Fig. 2.5) incorporates three sensors - an ITG-3200 MEMS triple-axis gyro, an ADXL345 triple-axis accelerometer, and a HMC5883L triple-axis magnetometer - providing a total of nine degrees of inertial measurement. All of the sensors' outputs are processed by an on-board ATmega328 microcontroller and then output via RS232 serial communication [23].

Technical specifications:

- 3DOF gyroscope - ITG3200 [24]:
  - Measures up to ± 2000°/s;
  - 16 bit resolution;
  - operating frequency of 30 kHz.
- 3DOF accelerometer - ADXL345 [25]:
  - Measures up to ± 16g;
  - 13 bit resolution;
  - Operating frequency of 3.2 kHz.
- 3DOF magnetometer - HMC5883L [26]:
  - operating until ± 8 gauss;
  - 12 bit resolution;
  - operating frequency of 400 kHz.
- ATmega328 incorporated micro-controller;
- RS232 communication;
- Dimensions: 28 x 41 mm;
- 3.5-16 VDC input.

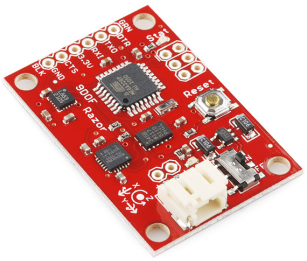


Figure 2.5: A RAZOR 9DOF - SEN 10736 IMU. Taken from [23].

Pololu MinIMU-9 v2

The Pololu MinIMU-9 v2 (Fig. 2.6) is an even more compact IMU which contains an L3GD20 3-axis gyroscope and a LSM303DLHC 3-axis accelerometer and magnetometer. The outputs of these sensors can be obtained via an I^2C interface and there are a total of 8 of these modules distributed along PHUA's body, in order to obtain the orientation of each main body part separately [27].

Technical specifications:

- 3DOF gyroscope - L3GD20 [28]:
  operating until ± 2000/s;
  16 bit resolution;
  operating frequency of 0.76 kHz.
- 3DOF accelerometer -LSM303DLHC [29]:
  operating until ± 16g;
  12 bit resolution;
  operating frequency of 400 kHz.
- 3DOF magnetometer - LSM303DLHC [29]:
  operating until ± 8 gauss;
  12 bit resolution;
  operating frequency of 400 kHz.
- I^2C communication.
- Dimensions: 20.32 x 12.7 mm;
- 2.5-5.5 VDC input.

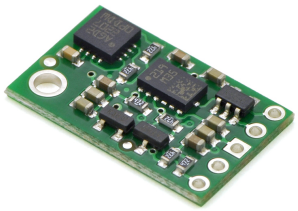


Figure 2.6: A Pololu MinIMU-9 v2 IMU. Taken from [27].

2.1.5 FANUC M-6iB/6S robotic manipulator

In order to perform experiments with high repeatability and great precision, a robotic manipulator present in LAR was used. The manipulator used is a FANUC M-6iB/6S (Fig. 2.7) and is part of the M-6iB series, comprised of small robots which are designed to approximate the reach of an operator. The variant present in LAR (6S) features six rotational joints (Fig. 2.8), each of them providing a different DoF, up to 6 kg of payload at wrist, 951 mm of horizontal reach and a ±0.08 mm repeatability interval [30].

Figure 2.7: A FANUC M-6iB/6S robotic manipulator. Taken from [31].

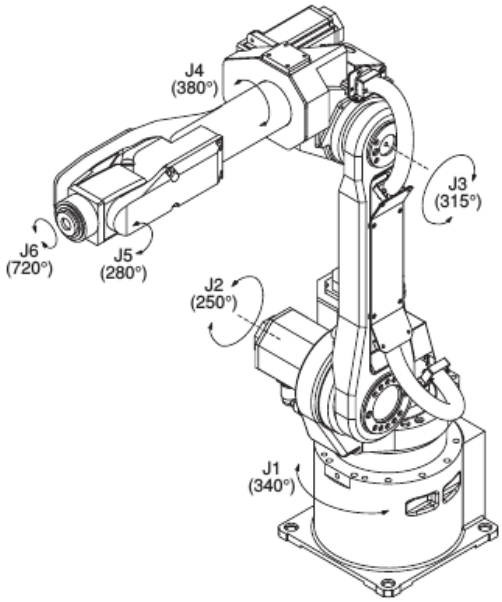


Figure 2.8: A schematic representing the 6 rotational joints of the FANUC M-6iB/6S. Taken from [30].

More details about the communication between the computer controlling the experiments and the robotic manipulator are present in section 2.2.6.

2.2 Software

2.2.1 Development platform and programming languages

All of this dissertation's work has been developed under the Ubuntu 16.04 LTS Open Source Operative System, a Linux distribution based on the Debian architecture [32] and most of the programming was done using the C and C++ programming languages.

2.2.2 ROS - Robot Operating System

The Robot Operating System (ROS) is an open source framework for writing robot software created in 2007. Through its collection of tools, libraries, conventions and even built-in applications, it aims to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms [33].

This allows for research groups around the world to participate in collaborative projects, which are all built on the same environment, making the resulting applications more robust. Applications created in projects can be packaged and used by other ROS users, allowing the community to save time costs developing what has already been done. ROS applications can even be written in different programming languages, such as C++, Python and Lisp, with experimental libraries being developed in Java and Lua [34].

It is also a distributed and modular framework, which means that users can use as much or as little of ROS as they desire [35].

What follows is a quick explanation of some keywords and concepts associated with this framework, followed by a simple diagram (Fig. 2.9) which will be useful when trying to understand the rest of this work [34]:

- **Packages**

  Packages are the main units for organizing software in ROS. It is the smallest thing you can build and release and in it there can be ROS runtime processes (*nodes*), ROS-dependent libraries, datasets, configuration files, launch files, message files, or anything else that is usefully organized together.

- **Metapackages**

  Metapackages are a specialized type of Package which only serve to represent a group of related other packages.

- **Nodes**

  Nodes in ROS are the equivalent of processes in normal operating systems. As was stated before, ROS is designed to be as modular as possible; Considering the example of a typical robot control system, it is made of many nodes. One node may control a laser range-finder, while another controls the wheel motors, another performs localization, another performs path planning, and so on. A ROS node is written with the use of a ROS client library, such as roscpp (for C++) or rospy (for Python).

- **Messages**

  Nodes communicate with each other by publishing messages to topics. A message is a simple data structure, which is comprised of fields with certain data types.

  msg files are simple text files for specifying the data structure of a message sent in ROS.

- **Topics**

  In ROS, messages are routed via a transport system which uses publishing/subscribing semantics. Nodes **publish** messages into a given topic. The topic is simply a name that is used to identify the content of the message. A node which is interested in a certain kind of data may **subscribe** to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics.

- **Bags**

  Bags are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.

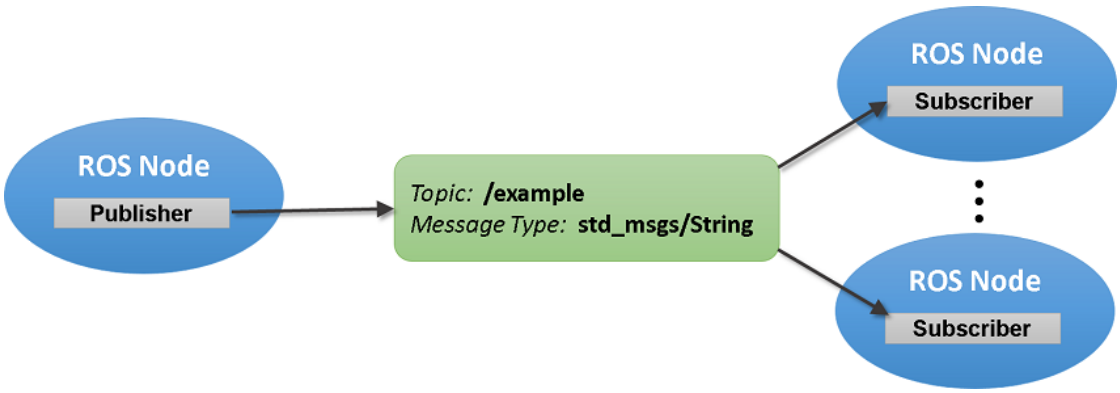


Figure 2.9: Two nodes subscribe the `/example` topic which contains a message of the type `std_msgs/String` and was published by a third node. Taken from [36].

2.2.3 OpenCV

Open Source Computer Vision Library (OpenCV) is an open source computer vision and machine learning software library, first released to the public in 2000. It is written on the C++ language but can work with C++, C, Python, Java and MATLAB and supports Windows, Linux, Android and Mac OS. It contains functions and algorithms that can help in many applications, like for example detecting and recognizing faces, identifying objects, tracking camera movements and moving objects, extracting 3D models of objects, and much more [37].

OpenCV has a modular structure, which means that it includes several shared or static libraries. The following modules are available: `core, imgproc, video, calib3d, features2d, objdetect, highui, gpu` and some other helper modules [38].

The modules used in this work were the following:

`core` module

A compact module defining basic data structures, including the dense multi-dimensional array `Mat` and basic functions used by all other modules.

`imgproc` module

An image processing module that includes linear and non-linear image filtering, geometrical image transformations (resize, affine and perspective warping, generic table-based remapping), color space conversion, histograms, and so on.

`calib3d` module

Basic multiple-view geometry algorithms, single and stereo camera calibration, object pose estimation, stereo correspondence algorithms, and elements of 3D reconstruction.

`highui` module

An easy-to-use interface for video capturing, image and video codecs, as well as simple UI capabilities.

2.2.4 ViSP - Visual Servoing Platform

The Visual Servoing Platform (ViSP) is a modular cross platform C++ library, which allows the prototyping and developing of applications using visual tracking and servoing techniques, and was developed by the Inria Lagadic team. With this library, it is possible to compute control laws that can be applied to robotic systems. It provides a set of visual features that can be tracked using real time image processing or computer vision algorithms [39].

It is divided into several modules as can be seen in figure 2.10. The modules which were most used in this work were the **`me`** (moving edges) and **`vision`** modules, which are described below [40].

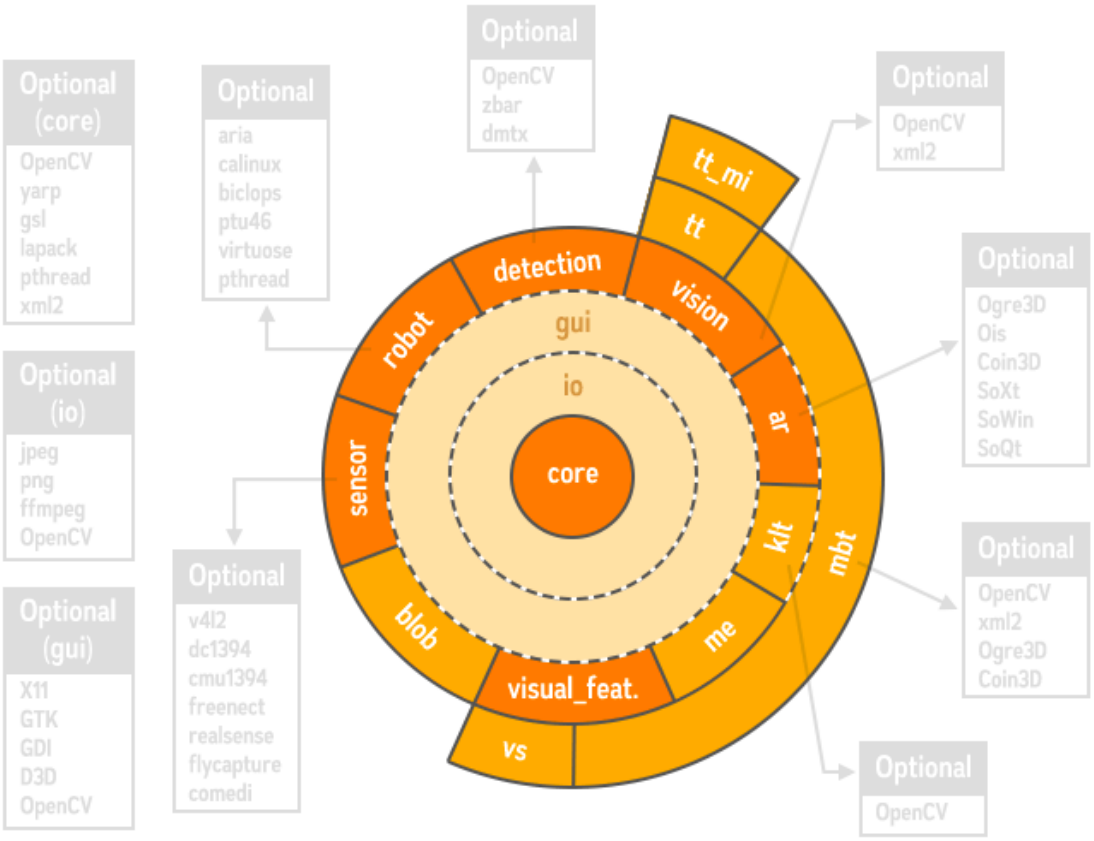


Figure 2.10: The ViSP software architecture. Taken from [40].

`me` module

This module provides real-time tracking of points normal to the object's contours. It allows for the tracking of lines and ellipses in pre-recorded video, or even a live feed coming from a video camera [41]. The first attempt at estimating verticality through visual data was made using this module at its base.

`vision` module

This module is able to compute a pose estimation (transformation matrix) or an homography from points using a robust scheme. For this work the feature that was used was the pose estimation feature [42]. By giving at least four points from an image plane and their corresponding 3D coordinates, ViSP is able to estimate the relative pose between the camera and the object frame. This pose is returned as an homogeneous transformation matrix [43]. An application using this module was the second method used to estimate verticality through visual data.

2.2.5 Arduino IDE and language

The Arduino IDE (Fig. 2.11) was the software used to write and upload code to the Arduino UNO R3 board. The Arduino language is merely a set of C/C++ functions

that can be called from the code. It basically acts as a simplified language adequate to all levels of programmers. In every Arduino program there are two functions. The `setup()` function is called when a program starts. It is used to initialize variables, pin modes, start using libraries, etc. It will only run once, after powering up the board or pressing the reset button. There is also a `loop()` function which continuously runs the code inside it in each loop cycle.

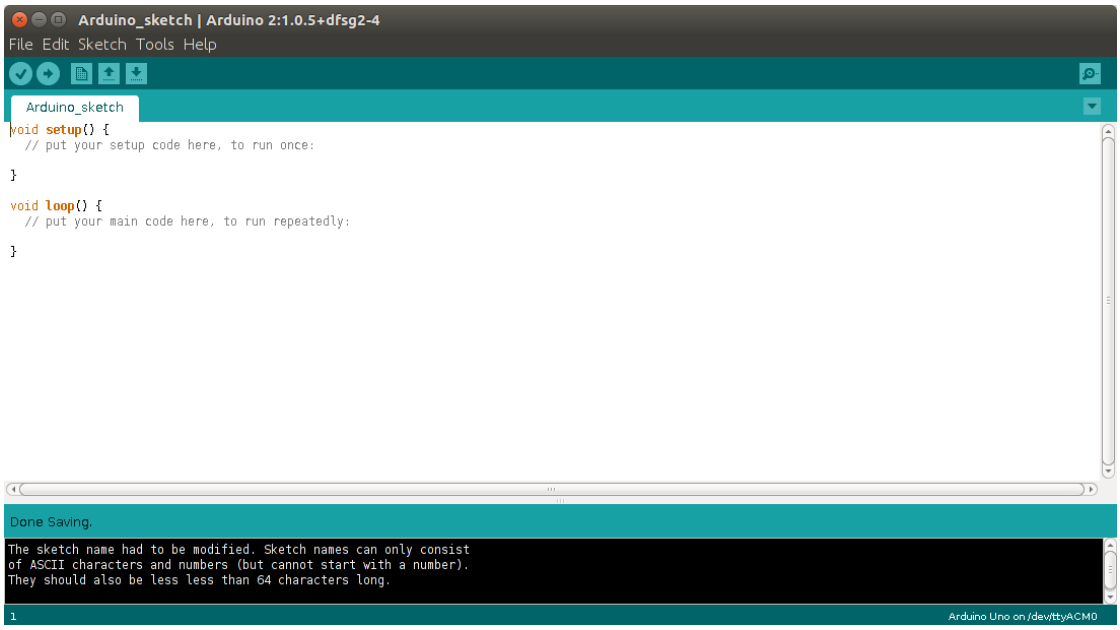


Figure 2.11: A screenshot of the Arduino IDE window.

Like other languages, the Arduino language can be extended by a number of libraries. The library that was primarily exploited in this work was the `Servo` library [44]. With it, it is simple to set a servomotor's shaft position and speed through the PWM outputs on the Arduino board. However, it is not possible to read a servomotor's current position through the use of this library, as was thought to be possible at the beginning of this work.

2.2.6 robCOMM - A Robot-Computer Interface

In order to control the FANUC M-6iB/6S manipulator remotely from a computer via the TCP/IP protocol, the robCOMM server present in the manipulator's controller was used. This server allows the user to send and receive information between both devices. This communication is made using a language developed in LAR, called robCOMM [45], which for example allows the user to move the robotic manipulator to a given set of joint/cartesian coordinates, read the current values of each joint, etc.

For a full list and explanation of the instructions available in this language, please refer to [45] and [46]. The instructions used in this dissertation were the RUNTPP and GETCRJPOS functions, which are described below.

RUNTPP function

This function allows a remote device to command the manipulator to run a previously created program which is present in the controller's memory.

Syntax: RUNTPP<CR>tpProgramName<CR>

where <CR> stands for the carriage return character and tpProgramName is the name of the program the user wishes to execute.

This function returns either "0<CR>" if unsuccessful or "1<CR>" if successful.

GETCRJPOS function

This function returns the current joint configuration of the robotic manipulator to the remote device.

Syntax: GETCRJPOS

This function returns <CR>...<CR> where "..." are the joint values followed by either "0<CR>" if unsuccessful or "1<CR>" if successful.

2.2.7 Glade and GTK+ toolkit

Glade is a Rapid-Application Development (RAD) tool to enable quick & easy development of user interfaces for the GTK+ toolkit and the GNOME desktop environment. These interfaces are saved as an Extensible Markup Language (XML) file, and are dynamically loaded by applications using the GtkBuilder GTK+ object. By doing so, Glade XML files are programming-language independent and the interfaces they contain are generated at runtime [47].

In the semester previous to the one this work took place in, there was an assignment for the *"Projecto em Automação e Robótica"* class whose purpose was the familiarization with the hardware used in this dissertation. The assignment was to create a simple graphical interface (Fig. 2.12) which would allow a user to control the position of the servomotors in the already existing PTU and visualize the image obtained with the camera placed on that said PTU.

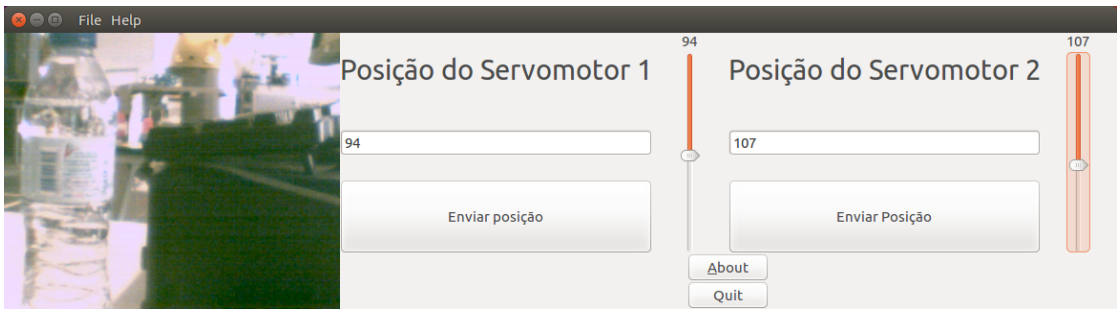


Figure 2.12: A screenshot of the created user interface.

As such, Glade and the GTK+ toolkit were the tools used to create this interface, along with the Arduino code which controlled the servomotors and code in the C language

which was used to mainly handle the interface's callbacks and to send the user's desired motor position to the Arduino board.

Chapter 3

Proposed solution

This chapter describes the work that was specifically developed to fulfill this dissertation's objectives, which include hardware modifications made to the existing PTU and the tools and methods used to estimate verticality and then stabilize the PTU (humanoid head).

3.1 Hardware modifications to the PTU

The existing PTU was built with the purpose to track a given target when the relative position between the PTU and the target changes [8]. However, this objectives of this work are different. After being able to estimate verticality using either visual or inertial data, the goal is to stabilize the unit where the camera is mounted on, so that the obtained image is aligned with the gravity vector (the ground would have to be parallel to the x-axis of the image). As such, modifications to the existing PTU had to be made to turn it into a Roll-Tilt Unit (RTU), since the degree of freedom associated with the pan angle is irrelevant when trying to balance a humanoid robot.

By making the roll and tilt angles variable, it would then be possible to send commands to the servomotors present in the RTU so that the camera (and the image it records) would be aligned with the gravity vector.

Therefore, a structure made of aluminum was designed in order to change the degrees of freedom of the unit where the servomotors were mounted. This part had to fix itself on the previously designed parts and it was made to be as light and as short as possible, so that the torque the servomotors had to apply would be minimum. The camera's optical axis was also made to intersect both of the servomotor's axis (see Fig. 3.1), so that when the motors apply a rotation, the camera would stay in the same place. The technical drawings associated with this alteration are present in Appendix A (PTU to RTU transformation section).

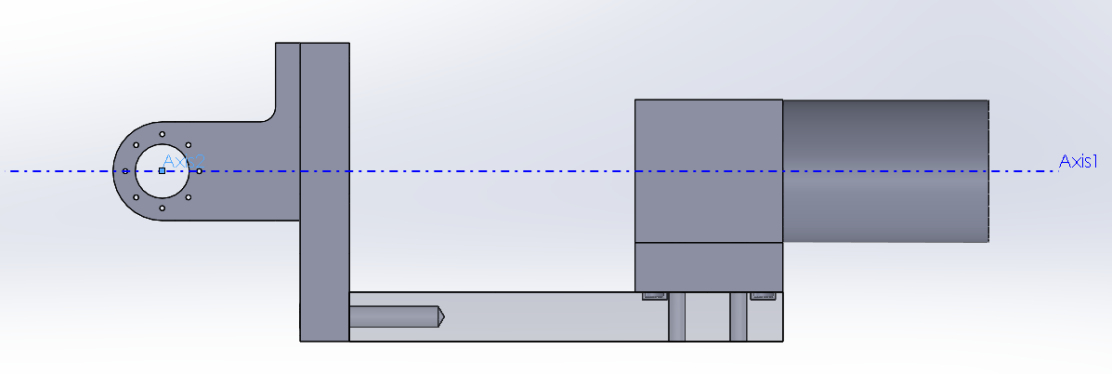


Figure 3.1: Visual representation of the axes in the RTU. "Axis1" corresponds to the roll servo axis and the camera's optical axis and "Axis2" corresponds to the tilt servo axis.

It is important to note that from this point on in the dissertation, the unit where the camera is mounted will now be called a RTU instead of a PTU. Besides this transformation from a PTU to RTU, some of the mechanical parts had to be replaced, because some screw threads were deteriorated from excessive use. To rebuild these parts, the drawings on Appendix A of [8] were used.

3.2 ROS packages

The developed software was organized in different ROS packages, so that if another work needs to only use a specific part of the solution, it is possible to do it. Note that many ROS packages used in this dissertation are needed for the remaining packages to work and are therefore considered dependencies.

What follows is a list of the packages, and a brief explanation of the nodes developed specifically by the author of this work:

`head_stabilization` package

This package contains the `motor_control` node which subscribes a ROS topic with the measured error in position and performs calculations based on a Proportional-Derivative (PD) control to command the RTU's servomotors to a certain angular position, with a given angular speed. It also reads the motors' current position and publishes it to a ROS topic.

The package also contains the `pose_estimation` node, which subscribes to ROS topics containing coordinates of a chessboard square's corners, if there is a chessboard present in the camera image. It then calculates the geometrical transformation between the camera in the RTU and a chessboard present in the camera's image, allowing the user to know the chessboard's roll and tilt angles relative to the camera.

`image_converter` package

This package contains the `image_converter` node which subscribes to an image message and performs an undistortion of the image, based on the parameters found

when previously calibrating the camera. It also detects the presence of a chessboard in the image and calculates the chessboard square's corner positions. It then publishes some of these point coordinates into different ROS topics, which can then be used by other nodes to, for example, initialize tracking operations.

`movingedges` package

This package contains the `movingedges` node which also subscribes to ROS topics containing the coordinates of a chessboard which is present on the camera image. It then performs the tracking of a line made by the connection of these points, allowing the user to know the chessboard's roll angle. It then publishes this angle into another ROS topic.

3.3 Analyzing an image to estimate verticality

After some time was spent studying the problem at hand it was possible to conclude that there are mainly two ways to estimate verticality through an image.

The first way implies doing image analysis when nothing is known about the objects in the scene. However, this comes with a problem: if there are no special features to analyze in the image (e.g. the robot is staring at a blank wall), then there is no way to actually perform an estimation of verticality. In the previous example, the robot may even be falling to its side, but since there is nothing worth analyzing on the image captured by the camera, one would have to resort to inertial data in order to prevent the robot's fall.

The second way implies having a known object present in the image and also to know the object orientation in relation to the environment. This was the way explored in this dissertation and in order to do so a chessboard previously used to perform camera calibrations was the object used (Fig. 3.2).

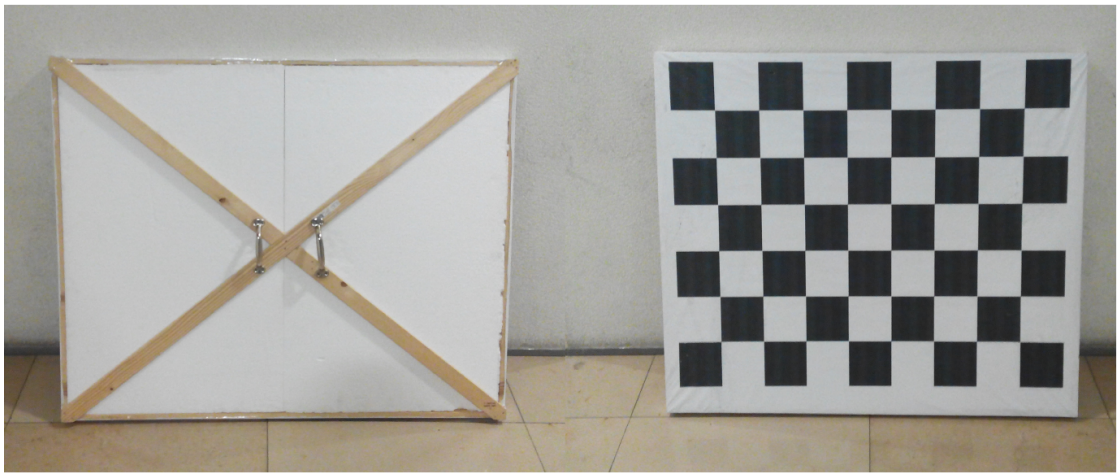


Figure 3.2: The chessboard used as a known object. Taken from [48].

If, for example, the chessboard is aligned with the ground in the real world, then it is easy to also know the camera orientation in relation to the ground. To do it, one would

just have to obtain the chessboard orientation in relation to the camera by analyzing the image, and the camera orientation to the ground would be that same value.

In order to obtain the chessboard orientation, two methods were used (section 3.5). This is because the first method would only be able to obtain the roll orientation while the second method made it possible to obtain both the roll and tilt orientation of the chessboard, and therefore the camera roll and tilt orientation too.

3.4 Image acquisition and preprocessing

In order to obtain images from the camera in a format which is transferable in the ROS framework, a node built in a previous work [49] was used. This node is called `Point_Grey` and its source file is `pointgrey_fl3-ge-28s4-c_driver.cpp` located in the `pointgrey_fl3_ge_28s4_c` ROS package.

This node also configures the camera's parameters, therefore allowing the user to choose the desired image format. The parameters used in the work where this node was created also fit the needs of this dissertation, so they were kept the same. As such, the video mode used was mode 1 with a 964x728 resolution which allows for a maximum of 14 frames per second (FPS) [50](page 77).

After retrieving the next image frame into an OpenCV image, this node converts it to a ROS image message using the `cv_bridge` ROS package (Fig. 3.3). The resulting image message is published on a topic named `/RawImage`, which can then be used by other nodes to acess the image and analyze it.

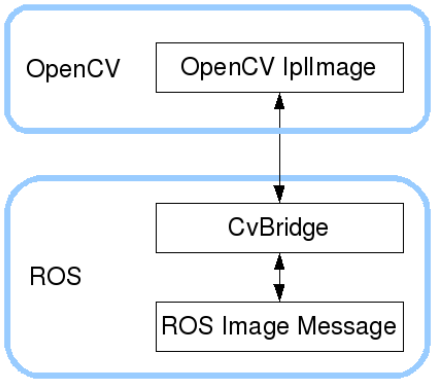


Figure 3.3: `cv_bridge` application example. Taken from [51].

Since both of the methods used to estimate verticality need some information of where the chessboard is in the image in order to be initialized, the image processing needed to obtain the chessboard corners is performed on the `image_converter` node, described next.

`image_converter` node

The source file of this node is `image_converter.cpp` and it is located in the `image_converter` ROS package. Its main function is to provide other nodes with the coordinates of the internal chessboard corners present in an image so they can begin

tracking (or other functions). In order to do so, it subscribes to the previously mentioned `/RawImage` topic, which contains the image captured by the camera in the ROS image message format. Then it converts that image message back into an actual OpenCV image, using the `cv_bridge` ROS package once again. Firstly, the image was undistorted based on the intrinsic parameters and distortion coefficients of the camera. To obtain these values, the camera was calibrated using the `camera_calibration` ROS package (Fig. 3.4). In order to perform this calibration, the tutorial found in [52] was used. The results of the calibration are shown in Table 3.1;

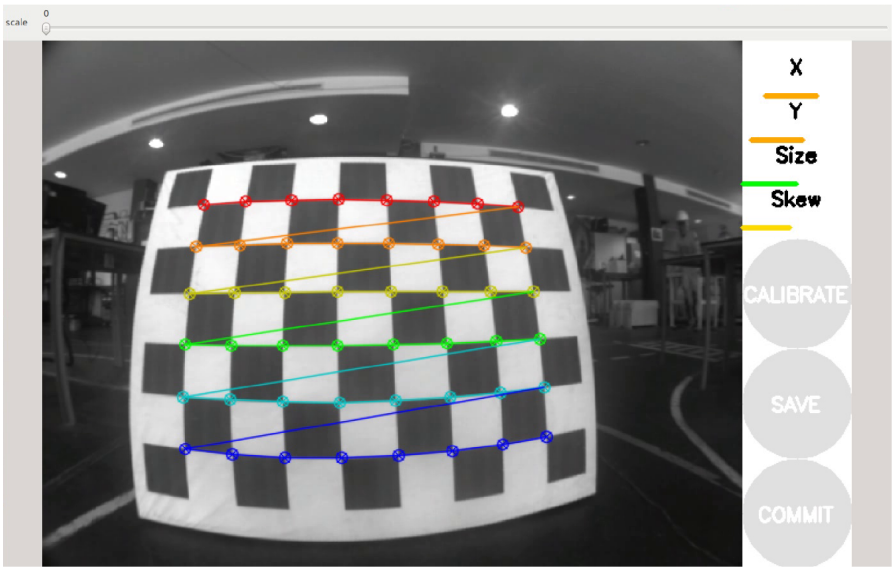


Figure 3.4: A screenshot of the camera calibration process. Taken from [49]

Table 3.1: Intrinsic parameters and distortion coefficients obtained from camera calibration.

Parameters	**Values (in pixels)**
f_x	503.406178
f_y	504.095921
c_x	488.641027
c_y	362.208972
Distortion coefficients	**Values**
k_1	-0.301046
k_2	0.069974
p_1	0.000667
p_2	-0.000580
k_3	0.000000

where f_x and f_y are the focal length in pixels;
c_x and c_y are the optical centers in pixels;
k_1, k_2 and k_3 are radial distortion coefficients;
and p_1 and p_2 are tangential distortion coefficients.

After removing the distortion with the `undistort()` OpenCV function, the image was converted from the RGB to the grayscale space so that the `findChessboardCorners()` OpenCV function could be used.

Syntax: bool **findChessboardCorners**(InputArray **image**, Size **patternSize**, OutputArray **corners**, int **flags**)

Parameters:
- **image** is an 8-bit grayscale or color image.;
- **patternSize** is the number of points per row and column of the chessboard in the CvSize format;
- **corners** is the output array where the corner coordinates are stored;
- **flags** is an integer which determines if certain optional operations are performed.

This function outputs a non-zero value if all the corners are found, and zero otherwise.

After using the `findChessboardCorners()` function, a simple representation of the most important points are displayed on the screen (Fig. 3.5).

The points are then published into separate topics, each topic containing the x and y coordinates of a corner. These topics contain a ROS message of the type `Num`, which was created specifically for this work. Its definition is made in the `Num.msg` file, located in the `msg` folder of the `image_converter` package. Messages of this type can contain two numbers of the built-in float64 type, which are equivalent to a double in the C++ language.

The process described above is repeated for every new frame that arrives from the `/RawImage` topic.

Figure 3.5: The chessboard points used to initialise both tracking methods.

3.5 ViSP tracking methods

3.5.1 Line Tracking method

The first method of verticality estimation consists of using the line tracking capabilities of the ViSP library. ViSP contains a tracker for moving edges which can track either lines or ellipses. To initiate line tracking, it is necessary to provide two points belonging to the line to be tracked. ViSP allows for these points to be provided manually with the user's interaction (selecting the two points with the mouse in a screenshot of the camera feed) or automatically by providing the coordinates in the image of these two points. The points used were points one and two present in Figure 3.5. These two points form a line in the chessboard (which is made of the sides of the chessboard squares) and the tracking itself is done on the `movingedges` node, described next.

`movingedges node`

The source file of this node is `movingedges.cpp` and it is located in the `movingedges` ROS package. Its main function is to track a line present in an image and publish that line orientation to be used by other nodes.

The tracking is made by first subscribing to the `/RawImage` topic and then initializing the moving edges parameters with the lines of code shown in code snippet 3.1.

Code snippet 3.1: Moving edges parameters initialization

```
vpMe me;
me.setRange(10);
me.setThreshold(45000);
me.setSampleStep(3);
int nbLines = 1;
vpMeLine line[nbLines];
```

where **me** is the moving edges object of the vpMe class;
setRange is a method of the vpMe class which sets the seek range (in pixels) on both sides of the tracked line;
setThreshold is a method of the vpMe class which determines if the moving edge is valid or not;
setSampleStep is a method of the vpMe class which sets the minimum distance (in pixels) between two line points.
nbLines is the number of lines to track;
line is the tracker object of the vpMeLine class;

The values used above were the ones that delivered the best results after testing a wide range of different values. After initializing the moving edges parameters, the tracker is initialized with the instructions presented in code snippet 3.2.

Code snippet 3.2: Line tracker initialization

```
line[0].setMe(&me);
line[0].setDisplay(vpMeSite::RANGE_RESULT);
do{
        cout << "Didn't find the chessboard points, retrying.." <<
            endl;
        usleep(1000000);
}while( (pt1.get_i()==0 && pt1.get_j()==0) && (pt2.get_i()==0 && pt2.
    get_j()==0) );

line[0].initTracking(I,pt1,pt2);

line[0].display(I, vpColor::red) ;
```

In code snippet 3.2, the first two lines associate the tracker object with the moving edges parameters and allow it to display additional information on the image. Afterwards, the node awaits the reception of the coordinates of the points used to initialize the tracking, since without them, the tracker object wouldn't start and the node would shutdown.

If the point coordinates are received, the tracking is initialized and the tracking results are displayed on the image with a red color (Fig. 3.6).

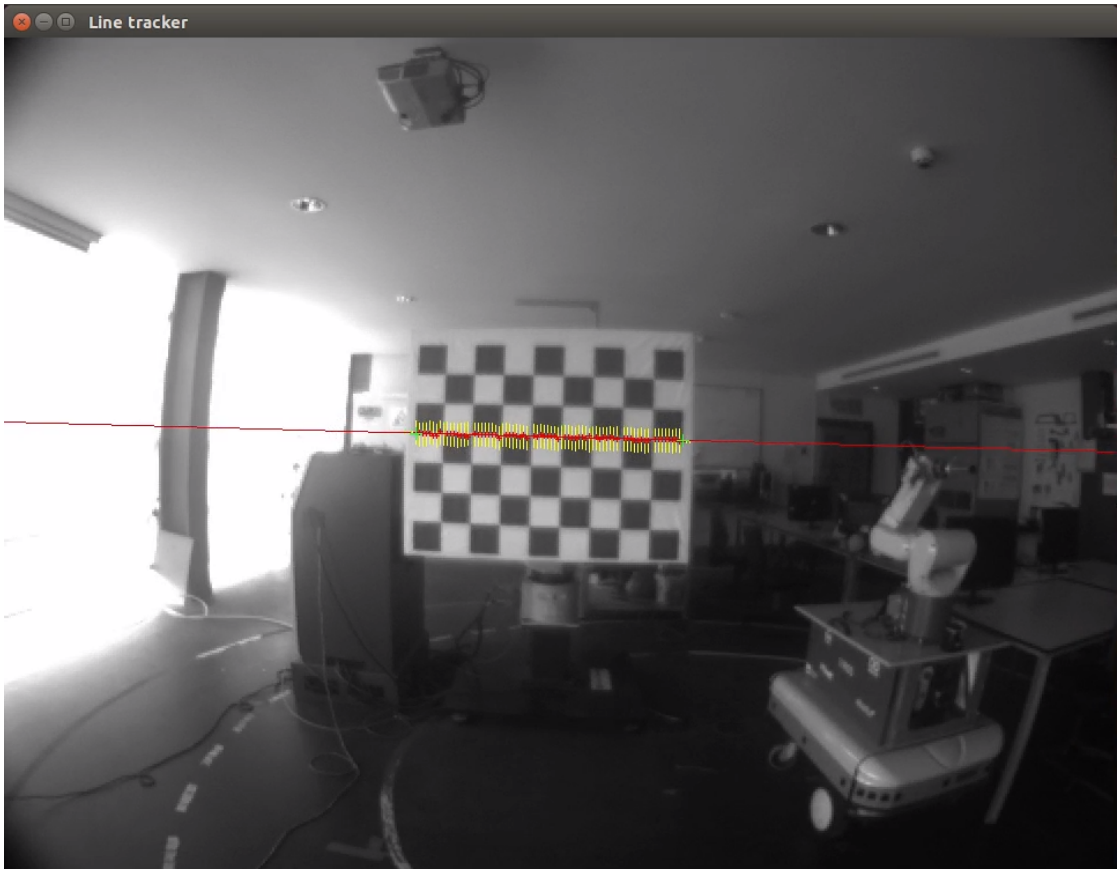


Figure 3.6: Results of the ViSP line tracking method.

The angle that the tracked red line does with the x-axis is equal (in absolute value) to the roll orientation of our camera. The roll angle is then published to a ROS topic named `/angle` which contains a message of the `Num` type, defined previously in the `image_converter` package.

3.5.2 Pose Estimation method

The second method of verticality estimation consists of using the pose estimation capabilities of the ViSP library. ViSP contains a module that can estimate the pose (i.e. geometrical transformation) of an object in relation to the camera which is recording the tracked object. To initiate pose estimation, it is necessary to provide four points that belong to the plane of the object and their real world coordinates as well. The points used were points three, four, five and six present in Figure 3.5. The method itself is applied on the `pose_estimation` node, described next.

`pose_estimation` node

The source file of this node is `pose_estimation.cpp` and it is located in the `head_stabilization` ROS package. Its main function is to estimate the pose of the chessboard present in LAR (since the 3D coordinates stored belong to this chessboard

in particular) in relation to the camera which is recording it. It then publishes that chessboard's roll and tilt angles into a ROS topic.

The pose estimation is made by first subscribing to the `/RawImage` topic and then initializing some variables, as shown in code snippet 3.3.

Code snippet 3.3: Variable initialization for pose estimation method

```
// Parameters of our camera
vpCameraParameters cam(503.406178, 504.095921, 488.641027, 362.208972)
    ;

// The pose container
vpHomogeneousMatrix cMo;

std::vector<vpPoint> point;
double W = 0.37;
double H = 0.27;
point.push_back( vpPoint(-W, -H, 0) );
point.push_back( vpPoint( W, -H, 0) );
point.push_back( vpPoint( W,  H, 0) );
point.push_back( vpPoint(-W,  H, 0) );
```

To perform a pose estimation, the intrinsic camera parameters are necessary, as well as inputting four real world coordinates of chessboard corners. The points that are input here must correspond to the points whose pixel coordinates are supplied by the `image_converter` node. Since the chessboard measures about 74 centimeters horizontally from point 3 to point 4 and about 54 centimeters vertically from point 3 to point 6, and we want the object reference frame to be in the center of the chessboard, the points were created as shown in code snippet 3.3. Point 3 (from Fig. 3.5) is going to correspond to point[0], whose coordinates are (-0.37,-0.27). Point 4 corresponds to point[1] and so on. An important thing to note is that points 3 and 4 have corresponding negative y-axis values because it is a common conception that the y-axis grows from the top to the bottom in an image.

The `cMo vpHomogeneousMatrix` object is the homogeneous matrix where the pose estimation is going to be stored. After these variable initializations, the `computePose()` function is called as shown in code snippet 3.4.

Code snippet 3.4: Call of the `computePose()` function

```
computePose(point, dot, cam, first_time, cMo);
```

The `computePose()` function is defined as shown in code snippet 3.5.

Code snippet 3.5: Definition of the `computePose()` function

```
void computePose(std::vector<vpPoint> &point, const std::vector<
    vpImagePoint> &dot, const vpCameraParameters &cam, bool first_time,
     vpHomogeneousMatrix &cMo){
        vpPose pose;     double x=0, y=0;
        for (unsigned int i=0; i < point.size(); i ++) {
                vpPixelMeterConversion::convertPoint(cam, dot[i], x, y
                    );
                point[i].set_x(x);
                point[i].set_y(y);
                pose.addPoint(point[i]);
        }

        if (first_time == true) {
                vpHomogeneousMatrix cMo_dem;
                vpHomogeneousMatrix cMo_lag;
                pose.computePose(vpPose::DEMENTHON, cMo_dem);
                pose.computePose(vpPose::LAGRANGE, cMo_lag);
                double residual_dem = pose.computeResidual(cMo_dem);
                double residual_lag = pose.computeResidual(cMo_lag);
                if (residual_dem < residual_lag)
                        cMo = cMo_dem;
                else
                        cMo = cMo_lag;
        }
        pose.computePose(vpPose::VIRTUAL_VS, cMo);
}
```

In short, this function converts each of the four points from pixel coordinates to meters by using `vpPixelMeterConversion::convertPoint()`, updates each points real world coordinates and adds them to the `pose` object of the `vpPose` class. After all points are added to the `pose` object, the pose is estimated, and is stored in the `cMo` homogeneous matrix. If it is the first time this function is called, a first estimation of the pose is made by two different approaches: the `vpPose::DEMENTHON` and the `vpPose::LAGRANGE` approaches. The approach which produces the best result, (measured by `vpPose::computeResidual()`) will be the one chosen as the first estimation.

After the pose estimation has been made, the results are displayed in the screen (Fig. 3.7) and the roll and tilt angles of the chessboard are extracted from the pose matrix and published on a ROS topic named `/angle` which contains a message of the `Num` type, defined previously in the `image_converter` package.

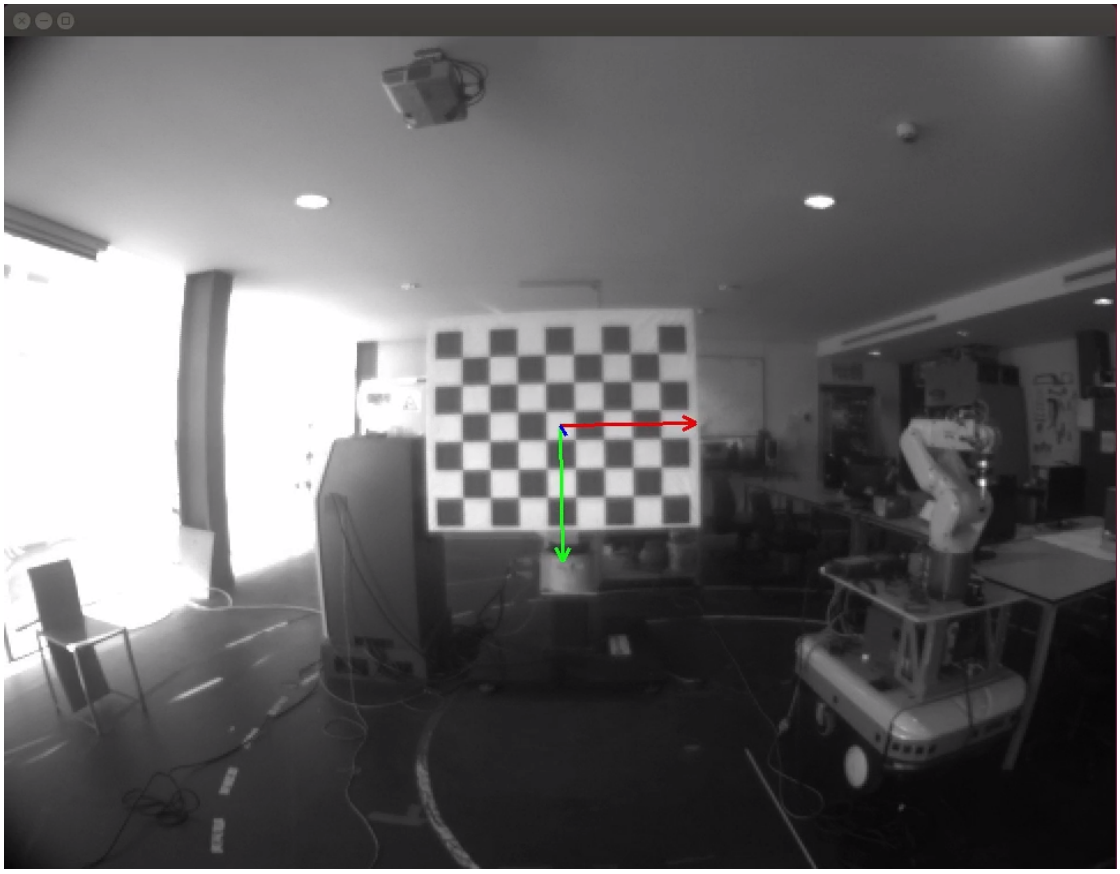


Figure 3.7: Results of the ViSP pose estimation method.

3.6 Stabilizing the RTU (Humanoid Head)

After obtaining an estimation of the roll and tilt camera angles in relation to the ground, it is necessary to stabilize the RTU using the mounted servomotors. The initial approach was to control both of the servomotors and read the inertial sensors through the same Arduino board. However, when implementing a solution with the Arduino Servo library, it was discovered that the `read()` function only returned the value of the last call to `write()`. As such, it doesn't return the actual current position of the servomotor, which is absolutely necessary for a robust control algorithm. Additionally, when receiving the inertial readings through serial communication, there seemed to be some spike in voltage in the PWM pins, which would make the servomotors jitter for an instant.

With the problems described above, it was decided to use the same infrastructure that was previously used to control the servomotors. All the details on how to connect each component to each other are described in section 3.1 of [53], while all the functions used to control the servomotors (e.g. read position, set position, read servo ID, set speed...) were developed by the same author in a ROS package named `hitec5980sg`, available in LAR toolkit v4 [54]. Two functions that are important for the understanding of this work will now be described:

Syntax: short unsigned int hitec_5980SG::**SetPosition**(int **id**,int **position**)

Parameters:
- **id** is the id of the servomotor;
- **position** is the position value to which the user wants to send the servo to (600-2400).

The `SetPosition()` function returns a response from the servo, if the message was well received, or 0xFFFF if an error ocurred.

Syntax: short unsigned int hitec_5980SG::**SetSpeedPosition**(int **id**,int **speed**)

Parameters:
- **id** is the id of the servomotor;
- **speed** is the speed value the user wants the servo to go with to the next position (0-255).

The `SetSpeedPosition()` function returns a response from the servo, if the message was well received, or 0xFFFF if an error ocurred. This response also includes the current position of the servomotor.

Before beginning to describe the node that implements the stabilization process itself, it is important to introduce some specific characteristics of these servomotors and their range of motion. The servomotor responsible for altering the roll angle has a full span of 180°. However, the functions developed in the `hitech5980sg` package only accept position values from 600 μs to 2400 μs (pulse width control). The 600 μs position corresponds to the -90° position and 2400 μs corresponds to +90° position. This means that a 1° increase signifies an increase of 10 units in the servomotors own unit system. Furthermore, the servomotor responsible for altering the tilt angle only has about 40° of range motion, counting from the +90° position, which means that its position values can range from 2000 μs to 2400 μs. This happens because of the way that the servomotor is installed in the RTU and reducing the position values any further would imply that some parts collide with each other, which is not admissible.

It was also intended that both servos could be operated at the same time, for stabilizing movements which were composed of both a roll and a tilt component at the same time. However, this proved impossible since with the current RTU setup, the roll servomotor also rotates the tilt servomotor. To better understand this problem, three representations of the camera in a scene with a global reference frame are shown in Figure 3.8. When the tilt servo is actuated and the roll servo is on a non-zero position, the camera's optical axis leaves the $y = 0$ plane, which is undesirable. As such, the stabilization process can only be done to one servo at a time.

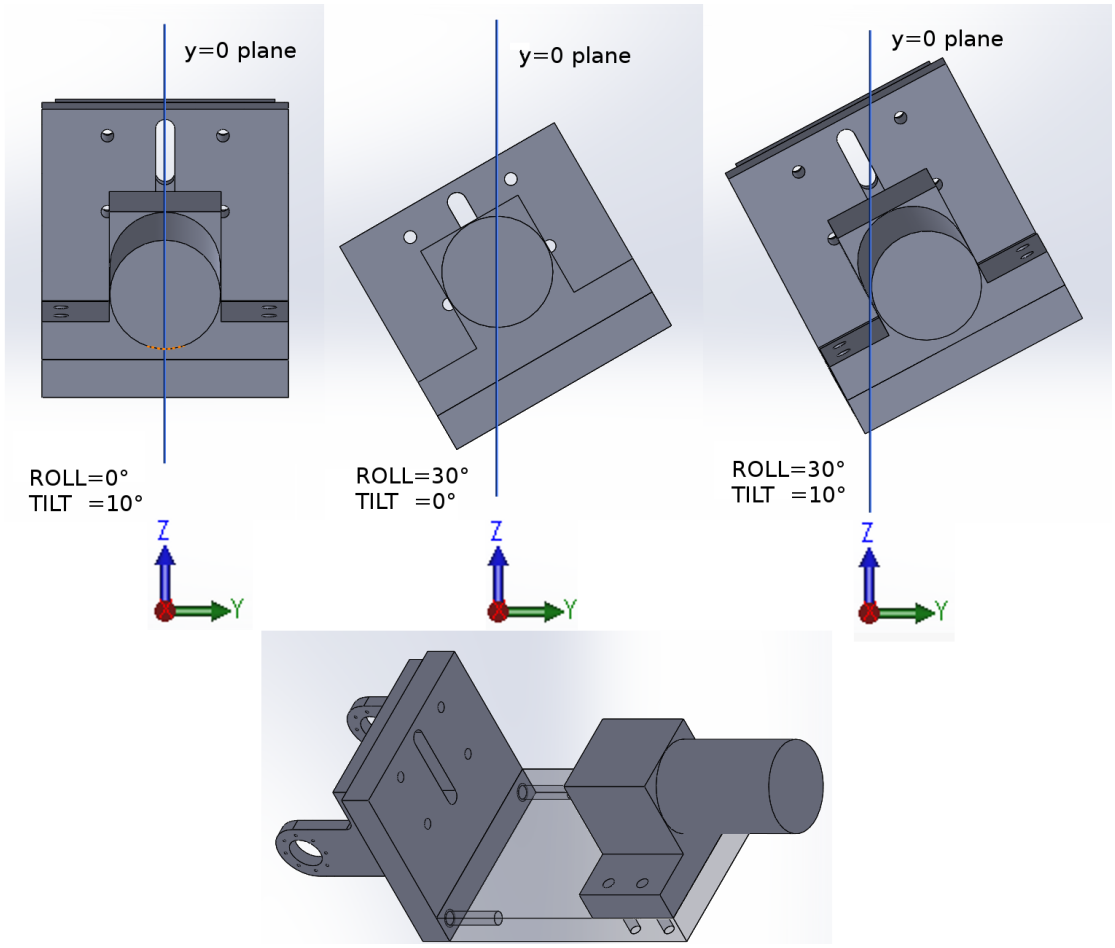


Figure 3.8: An explanation to why a change in both the roll and tilt angles makes the camera's optical axis leave the y=0 plane.

To further explain the stabilization process implemented in the `motor_control` node, it will be described next.

`motor_control` node

The source file of this node is `motor_control.cpp` and it is located in the `head_stabilization` ROS package. Its main function is to stabilize the RTU by controlling the servomotors' position and speed. It then publishes both of the servomotor's current positions into a ROS topic. It has a roll and a tilt mode, depending on the angle which is going to be stabilized.

A first version of this node implemented a control algorithm with just a proportional component. However, this proved to be insufficient as will be concluded in the experiments in section 4.2.2. Nevertheless, a short explanation of how it worked will be described in the following paragraph.

First, this node retrieves the roll and tilt angle errors from the `/angle` ROS topic, and then a calculation of the roll motor's position setpoint is done, as shown in equation

(3.1).

$$\theta^1_{\text{setpoint}} = \theta^1_{\text{error}} \times 10 + \theta^1_{\text{current}} \tag{3.1}$$

In equation (3.1), θ^1 stands for the position of the motor which is responsible for the roll angle stabilization. In an example where the current motor position is 1600 μs and the roll error is -10°, the resulting setpoint would be 1500 μs, which would mean that the servo was on its 10° position and the setpoint is now 0°. After this setpoint was calculated, if the setpoint was within admissible position values (600 μs to 2400 μs), the servomotor was simply sent to the resulting position via the `hitec_5980SG::SetPosition()` function.

This first control method worked well for slower motor speeds, but when setting motor speeds higher, overshoots would occur, resulting in general instability of the image (See section 4.2.2 for more details).

To solve these problems, a more refined control algorithm had to be implemented. As the final solution to the RTU stabilization problem, a PD control algorithm was used. Some small tests were made to see if an integrative component should be used, however it always resulted in a worse response from the system.

To determine the position setpoint for the servo responsible for the roll angle (using the roll mode), equation (3.2) was used.

$$\theta^1_{\text{setpoint}} = (\theta^1_{\text{Pterm}} + \theta^1_{\text{Dterm}}) \times 10 + \theta^1_{\text{current}} \tag{3.2}$$

In equation (3.2), θ^1_{Pterm} is calculated using equation (3.3) and θ^1_{Dterm} is calculated using equation (3.4).

$$\theta^1_{\text{Pterm}} = K_p \times \theta^1_{\text{error}} \tag{3.3}$$

$$\theta^1_{\text{Dterm}} = K_d \times (\theta^1_{\text{error}} - \theta^1_{\text{prev-error}}) \tag{3.4}$$

In equation (3.4), $\theta^1_{\text{prev-error}}$ is the roll error that happened in the previous iteration of the control sequence. K_p and K_d are the proportional and derivative coefficients, and their best values, after extensive testing (section 4.2.3), are $K_p = 0.7$, $K_d = 0.15$ when using the line tracking method and $K_p = 0.5$, $K_d = 0.4$ when using the pose estimation method.

The servo responsible for the roll angle was then sent to this setpoint, if the setpoint was within admissible position values (600 μs to 2400 μs), using the aforementioned `hitec_5980SG::SetPosition()` function, with its maximum speed set by the `hitec_5980SG::SetSpeedPosition()` function.

For the control of the servo which is responsible for the tilt angle, a PD control algorithm was also used, but this time the control variable would be the servomotor's speed and not its position. This is because when trying to apply equation (3.2) to the tilt servo, it would not always respond. A probable cause for this is the excessive torque it had to do in order to lift the camera, since the distance between the camera's center of gravity and the servo axis (moment arm) was too big, despite the efforts to minimize such distance.

The solution to this problem is sending the servo to one of its extreme positions (2000 μs or 2400 μs), depending on the sign of the tilt error, but with a varying velocity,

depending on the magnitude of this error. To determine the speed setpoint for the servo responsible for the tilt angle (using the tilt mode), equation (3.5) was used.

$$\dot{\theta}^2_{\text{setpoint}} = (\dot{\theta}^2_{\text{Pterm}} + \dot{\theta}^2_{\text{Dterm}}) \times 12.75 \tag{3.5}$$

In equation (3.5), $\dot{\theta}^2_{\text{Pterm}}$ is calculated using equation (3.6) and $\dot{\theta}^2_{\text{Dterm}}$ is calculated using equation (3.7). The 12.75 constant originates from the fact that it was intended that if the tilt error was equal to 20° the servo speed should be 255. Therefore, if $K_p = 1$, and $\dot{\theta}^2_{\text{Dterm}} = 0$ then $\dot{\theta}^2_{\text{setpoint}} = 255$.

$$\dot{\theta}^2_{\text{Pterm}} = K_p \times \theta^2_{\text{error}} \tag{3.6}$$

$$\dot{\theta}^2_{\text{Dterm}} = K_d \times (\theta^2_{\text{error}} - \theta^2_{\text{prev-error}}) \tag{3.7}$$

Since this is a totally different kind of control, the K_p and K_d are expected to change. Their best values, after extensive testing (section 4.2.3), are $K_p = 0.12$, $K_d = 0.1$.

After all the control algorithm steps are done, the node sends the current motor positions to a ROS topic named `current_motor_position` which contains a message of the `Num` type, defined previously in the `image_converter` package.

In order to change the mode in which this node operates, the user should to go to its source file and change the `mode` string variable's value to either "roll" or "tilt" and compile the package once again. The corresponding K_p and K_d must also be assigned. To make this task easier, a commented version of these changes is present on the file.

3.7 Controlling the robotic manipulator

To perform the experiments described in chapter 4, it was necessary to control the FANUC robotic manipulator. To do so, the `fanuc_control` ROS package, developed in [7] was used and slightly altered. This package integrates the robCOMM language (sec. 2.2.6) used to communicate with the manipulator's controller with the ROS framework. This makes it possible to publish the current value of the robot's joints to a ROS topic, which in turn allows the use of this data as an extremely precise ground truth for conducting experiments.

The only alteration made to the package was to insert the names of the console programs which contain the instructions to move the robot in each experiment into the `fanuc.cpp` file. When a different experiment is to be conducted, it is necessary to change the `tppname` variable's value accordingly and then compile the package again.

The `fanuc_control` node publishes the cartesian coordinates of the robot's end effector into the `fanuc_cart` ROS topic and the robot's joint values into the `fanuc_joint` ROS topic.

Chapter 4

Experiments and results

In this chapter, an explanation of the experiments conducted during this dissertation in order to evaluate the estimation of verticality and the stabilization of the RTU (humanoid head) will be presented, along with their respective results.

4.1 Using `rosbag` and `roslaunch`

In each experiment, data was recorded with a frequency of 100 Hz into a `.bag` file using the `rosbag` package [55] and then converted into a .csv file using a python executable named `bag2csv.py` [56].

In order to run these experiments in a more controlled fashion, two `.launch` files were created. These use the `roslaunch` tool [57] to launch the ROS nodes used in each experiment, as well as a `rosbag` node, which subscribes to all topics and records the data into `.bag` files. These files are named `chessboard_rosbag.launch` and `chessboard_v2_rosbag.launch` and can be found on the `launch` folder of the `head_stabilization` package.

4.2 Experiments with chessboard mounted on the manipulator arm

In this section, the experiments performed with the chessboard mounted on the manipulator arm will be described. After this description, the experiment results will be analyzed through the use of descriptive statistics (e.g. averages, standard deviations), plots and summary tables. Any unreasonable spikes in the servomotor position line of these plots is due to a malfunction of the servomotors when trying to read their current position and was not an actual position that the servo passed through.

This process of experiment description and subsequent result analysis will be repeated for each different experiment.

4.2.1 Line tracking evaluation

Experiment description

The first experiment was one that could evaluate the robustness of the line tracking method and that could also verify its viability when tracking objects travelling/rotating at high speeds. In these experiments, the chessboard had to be mounted on the manipulator and, to do so, more mechanical parts had to be designed and built. The drawings of these parts are in Appendix A (Chessboard connection to manipulator).

A Teach Pendant (TP) program named `JS_ROLL` containing movement instructions for the FANUC robotic manipulator was created (Fig. 4.1) in order to perform experiments where the roll angle would vary. These movements had to be able to demonstrate the aforementioned capabilities of the tracking method and are described next.

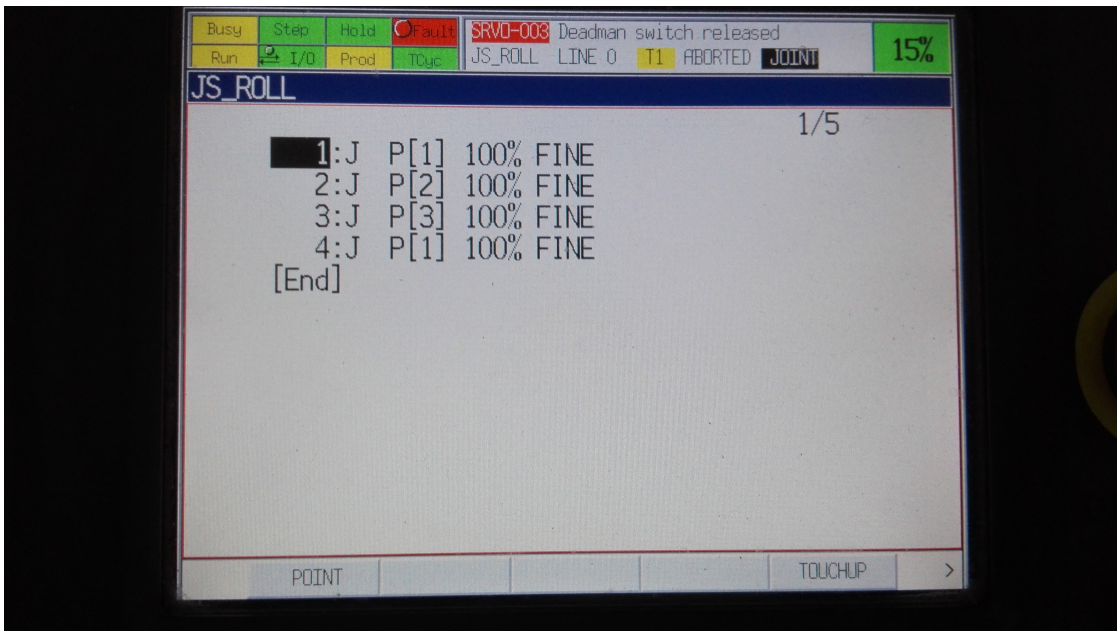


Figure 4.1: The `JS_ROLL` TP program.

The first line of the program commands the manipulator with the chessboard mounted on it to align its end-effector axis with the camera's optical axis, making the chessboard look like it is facing the camera and in an horizontal position (Fig. 4.2). The next line then applies a rotation to the sixth joint of the manipulator (See Fig. 2.8), so that a 45° clockwise rotation is observed. The third line then applies a 90° rotation on the same joint but in the opposite direction, so that the chessboard passes its initial position and rotates 45° counterclockwise. Finally, the manipulator is sent to its initial position and stops its movement.

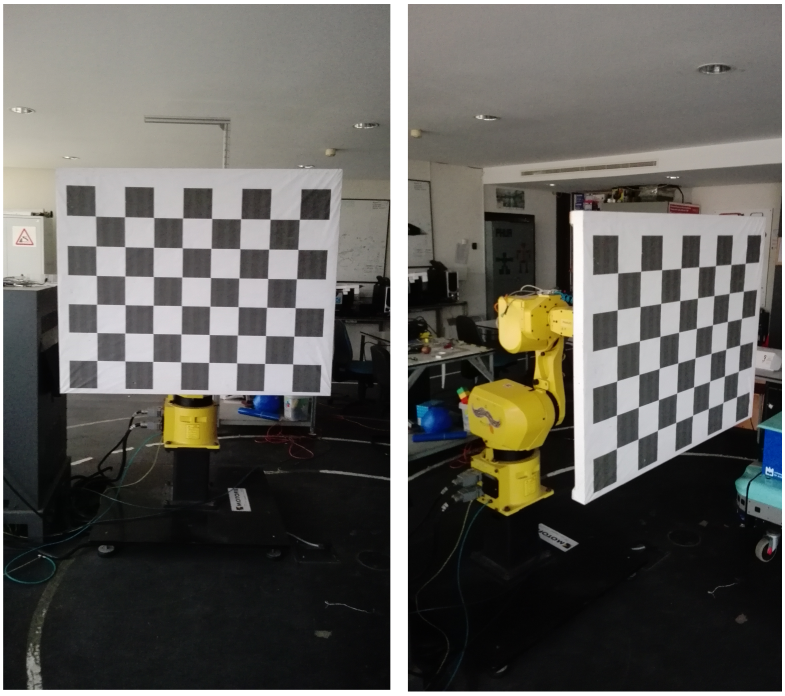


Figure 4.2: The initial position of the chessboard.

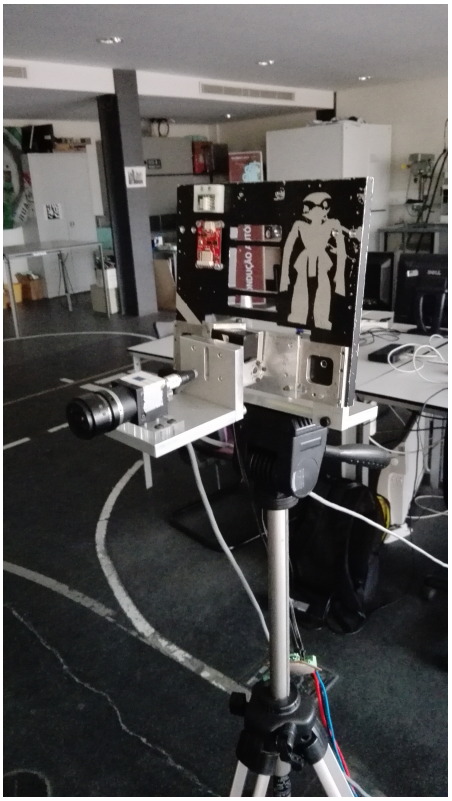


Figure 4.3: The RTU mounted in a tripod during experiments.

Whilst the manipulator is moving the chessboard, the RTU will be mounted on a tripod two meters in front of it (Fig. 4.3). The drawings of the mechanical parts designed to mount the RTU on the tripod are in Appendix A (RTU connection to tripod). No stabilization will occur yet, since the purpose of this experiment is to first verify if this is a viable method to track the chessboard. The range of motion applied to the chessboard also attempts to simulate the variation in the roll angle that the camera would observe when a humanoid robot is falling to its side. Several maximum speeds were applied to the manipulator's movements to test the limits of the line tracking method.

In total, 30 trials were made, divided equally between three different angular speed values. These values are expressed relative to the maximum joint velocity achievable by the manipulator and are the following: 5% (0.63 rad/s), 15% (1.88 rad/s) and 30% (3.77 rad/s).

Result analysis

To better demonstrate the results of this experiment, the plots of example trials run at each of the different manipulator speeds are shown in figures 4.4, 4.5 and 4.6 and a summary of these results is present in table 4.1. The error in these trials was considered to be the difference between the measured angle and the robot's joint position. The descriptive statistics that were gathered in these trials are the mean of the absolute value of the error ($\overline{E}$), the maximum error (E_{max}) and the standard deviation (σ) calculated using the population formula.

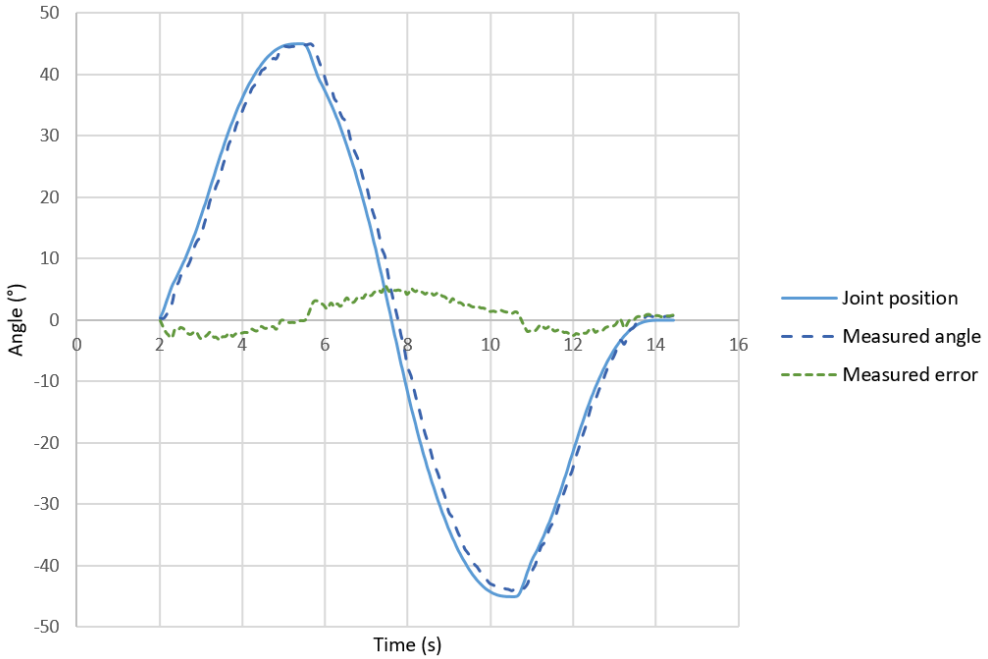


Figure 4.4: A plot of a trial run to evaluate line tracking by varying the roll angle. 5% manipulator speed used. $\overline{E} = 2.14°$, $E_{max} = 5.52°$ and $\sigma = 1.32°$.

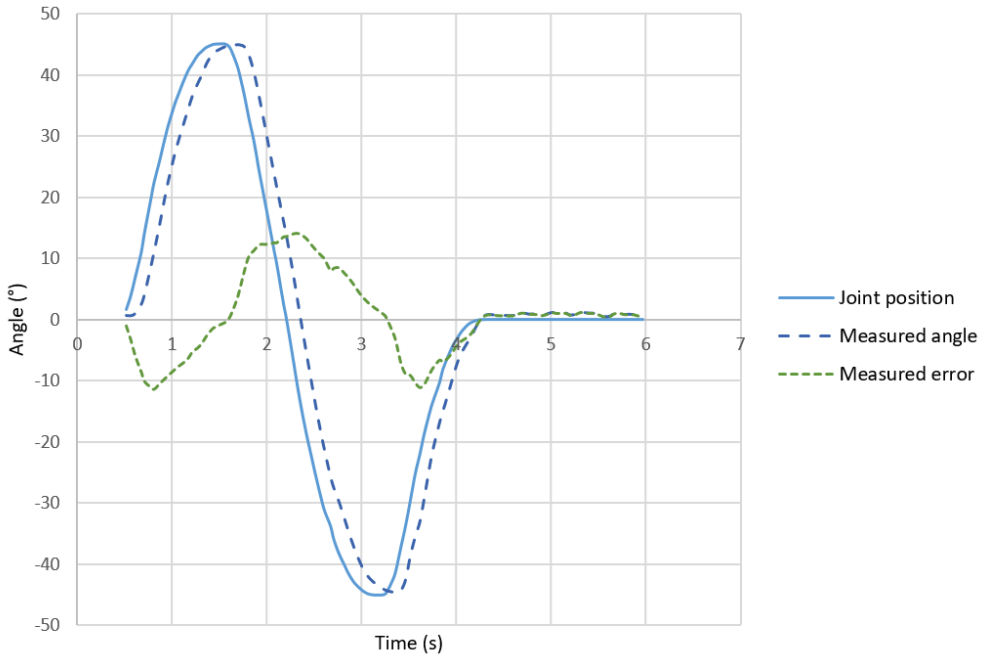


Figure 4.5: A plot of a trial run to evaluate line tracking by varying the roll angle. 15% manipulator speed used. $\overline{E} = 7.39°$, $E_{max} = 14.11°$ and $\sigma = 4.34°$.

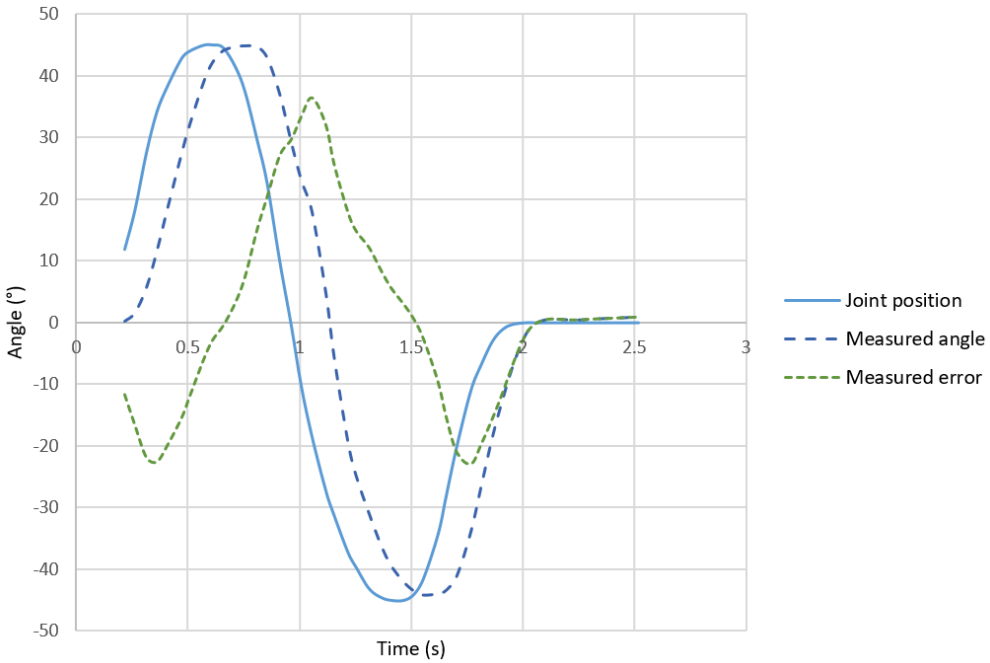


Figure 4.6: A plot of a trial run to evaluate line tracking by varying the roll angle. 30% manipulator speed used. $\overline{E} = 11.78°$, $E_{max} = 36.35°$ and $\sigma = 10.34°$.

Table 4.1: Line tracking evaluation results.

Descriptive statistics	**Manipulator speed**		
	5%	15%	30%
$\overline{E}$	2.14°	7.39°	11.78°
E_{max}	5.52°	14.11°	36.35°
σ	1.32°	4.34°	10.34°

By analyzing the data from the trials run at 5% manipulator speed, it is possible to conclude that the line tracking method works great at low angular speeds. When using greater manipulator speeds, however, the error starts to become larger, as can be seen in the trials run at 15% manipulator speed.

The fastest speed used in these trials was 30% of the manipulator's max speed. The results got even worse, indicating that the line tracking is indeed affected greatly by the speed of objects being tracked.

Overall, the line tracking method performed well since an angular speed of 3.77 rad/s (30% manipulator speed) is faster than what is thought to be needed for this work. A humanoid robot would never fall to its side that quickly unless if being pushed very hard. As such, it was assumed that a manipulator speed of 15% should be used for the rest of the experiments.

With a quick glance at each of the different plots we can also see that there is a "shift to the right" in the joint position line which increases because the delay between the chessboard moving and the tracking algorithm performing its calculations becomes more apparent. This delay is caused by the low frame rate of the camera mounted in the RTU, and is the main reason for the line tracking becoming worse at high angular speeds.

4.2.2 Proportional control of RTU

Experiment description

After being able to estimate verticality by using the line tracking method, an experiment whose purpose was to evaluate the quality of the control algorithm with just a proportional component (Sec. 3.6) was designed. It is very similar to the line tracking evaluation experiment, with the only difference being that the servomotor responsible for the roll angle in the RTU is now being actuated to "follow" the chessboard's movements.

To verify if the distance from the RTU to the chessboard had any influence in the results of the control algorithm, the tests were also repeated with a one meter distance between them.

In total, 30 trials were made, 15 of them with a two meter distance and the other 15 with a one meter distance. These 15 trials were divided as follows: five trials for a manipulator speed of 5% (0.63 rad/s) and a servomotor speed of 10/255 (actual physical value depends on servomotor load and was not calculated/measured); five trials for a manipulator speed of 15% (1.88 rad/s) and a servomotor speed of 10/255; and five trials for a manipulator speed of 15% (3.77 rad/s) and a servomotor speed of 32/255.

Result analysis

To better demonstrate the results of this experiment, the plots of example trials run at a 2 meter distance between RTU and chessboard are shown in figures 4.7, 4.8 and 4.9. Since the plots from trials run at a 1 meter distance are very similar, these will not be shown. A summary of these results is present in table 4.2. This time, the error in these trials was considered to be the difference between the servomotor position and the robot's joint position.

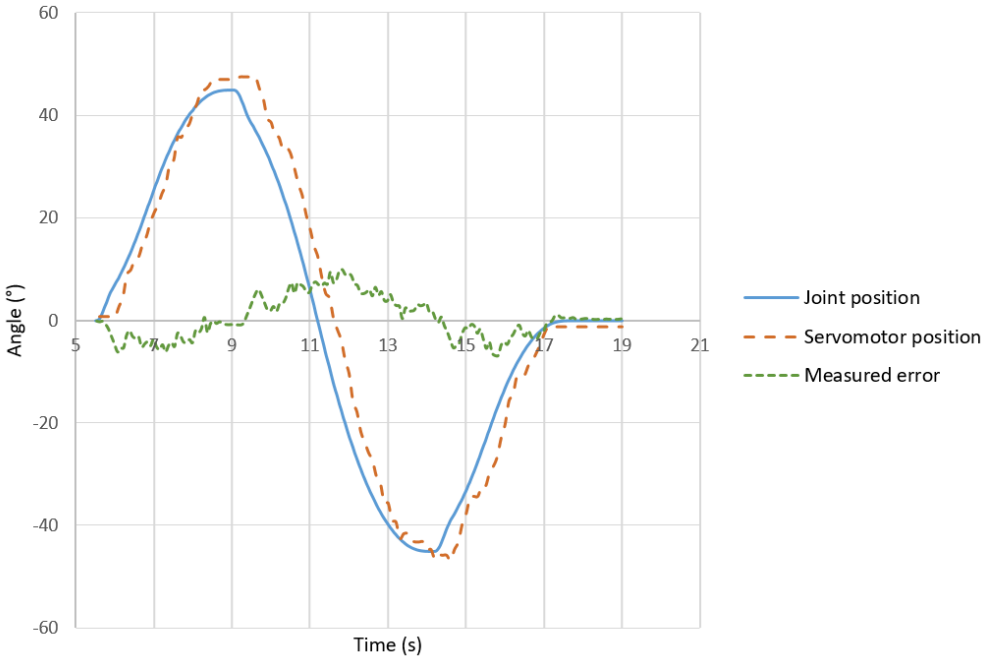


Figure 4.7: A plot of a trial run to evaluate proportional control using line tracking by varying the roll angle. 5% manipulator speed and 10/255 servo speed. 2 meters distance. $\overline{E} = 3.21°$, $E_{max} = 9.90°$ and $\sigma = 2.49°$.

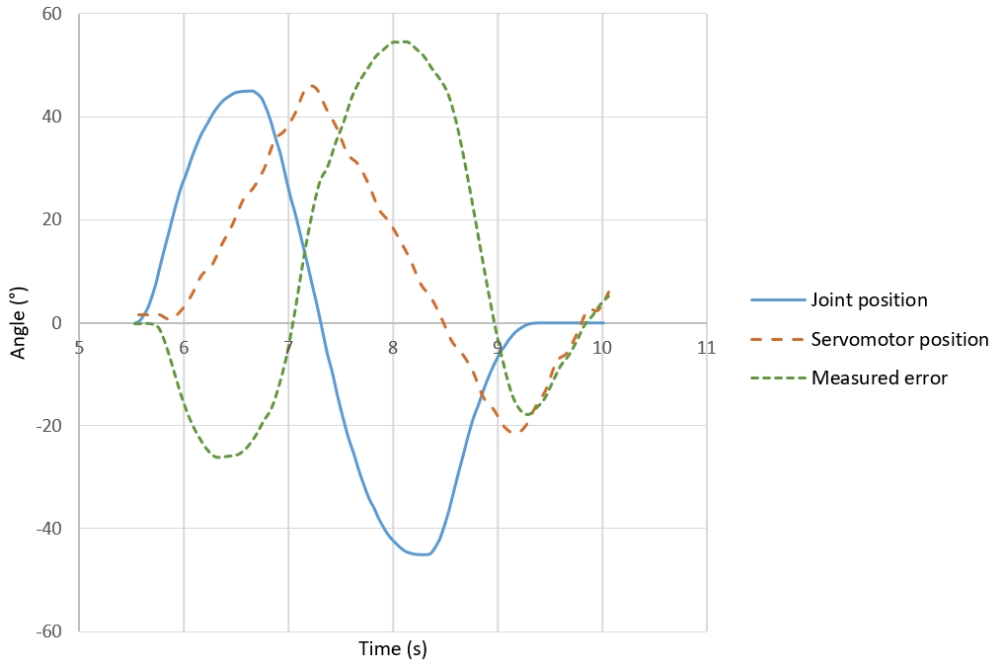


Figure 4.8: A plot of a trial run to evaluate proportional control using line tracking by varying the roll angle. 15% manipulator speed and 10/255 servo speed. 2 meters distance. $\overline{E} = 22.83°$, $E_{max} = 54.47°$ and $\sigma = 17.42°$.

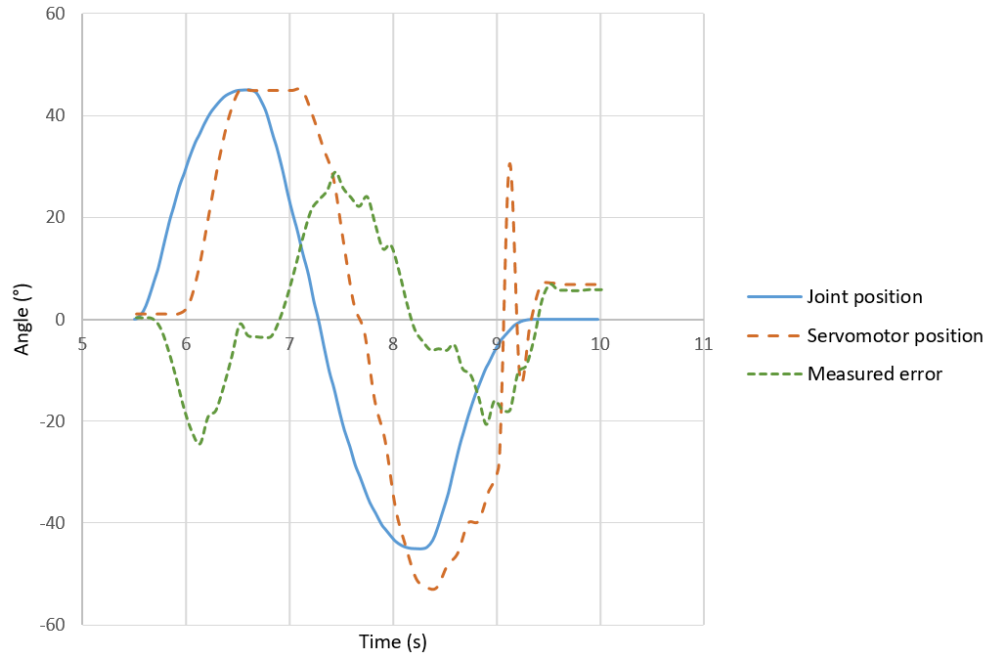


Figure 4.9: A plot of a trial run to evaluate proportional control using line tracking by varying the roll angle. 15% manipulator speed and 32/255 servo speed. 2 meters distance. $\overline{E} = 10.73°$, $E_{max} = 28.96°$ and $\sigma = 8.09°$.

Table 4.2: Proportional control of RTU results.

	Descriptive statistics	**Manipulator speed–Servo speed**		
		5%–10	15%–10	15%–32
1 meter	$\overline{E}$	3.45°	24.58°	13.83°
	E_{max}	9.80°	59.42°	36.68°
	σ	2.29°	19.50°	9.56°
2 meters	$\overline{E}$	3.21°	22.83°	10.73°
	E_{max}	9.90°	54.47°	28.96°
	σ	2.49°	17.42°	8.09°

From the trials run at 5% manipulator speed and 10/255 servomotor speed, it is possible to conclude that results were as expected when running at low angular speeds. When using a manipulator speed of 15%, whilst maintaining the servo speed at 10/255, the error increased dramatically, as was predicted, because the RTU could not keep up with the chessboard's movement because of its low motor speed. A servo speed of 32/255 was then used with the same manipulator speed to try and diminish this error. This specific value was used because as can be seen in Fig. 4.9, a servo speed this high begins to introduce instability to the system.

By analyzing table 4.2, it is also possible to conclude that there is not much difference between the results of experiments made at 1 meter distance and 2 meter distance. However, since 2 meter results were slightly better, this was the distance used for the remaining experiments.

After these trials, it was possible to conclude that a better alternative had to be found when it comes to the control algorithm used, and so work was done in order to implement a PD control algorithm.

4.2.3 PD control of RTU

Experiment description

After testing the limits of the proportional control algorithm, the PD control algorithm (Sec. 3.6) was also evaluated. For all experiments evaluating PD control, a servomotor speed of 255 (maximum speed) was used on roll angle experiments and the distance between RTU and chessboard was always two meters. For experiments where the roll angle was being tested, the manipulator maximum angular velocity used was 15% (108°/s) and for experiments where the tilt angle was being tested, a velocity of 5% (20°/s) was used.

Trials were made using both of the tracking methods described. To show the progressive tuning of the PD control's parameters when used in conjunction with the line tracking method, 15 trials were made with a manipulator speed of 15% (108°/s). Five trials were run with these parameters: $K_p = 0.3, K_d = 0$; five with $K_p = 0.7, K_d = 0.15$; and five with $K_p = 0.8, K_d = 0.15$. The same TP program (`JS_ROLL`) was used.

To better visualize the software architecture used during the trials with the line tracking method to stabilize the RTU, a diagram (Fig. 4.10) containing the relevant ROS nodes and topics was created using the `rqt_graph` tool [58].

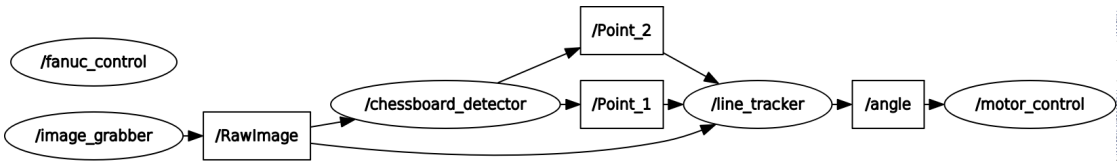


Figure 4.10: An overview of the ROS nodes and topics running during experiments on RTU stabilization using the line tracking method.

To show the progressive tuning of the PD control's parameters when used in conjunction with the pose estimation method, 15 trials were run when altering the chessboard's roll angle and 15 more when altering the chessboard's tilt angle. From the roll angle experiments, five trials were run with $K_p = 0.2, K_d = 0$; five with $K_p = 0.5, K_d = 0.4$; and five with $K_p = 0.7, K_d = 0.4$.

To perform experiments which test the control on the tilt angle, a second TP program was created named `JS_TILT` (Fig. 4.11).

Figure 4.11: The `JS_TILT` TP program.

The first line of the program sends the manipulator to the same initial position as in the other experiments. The second line applies a rotation to the manipulator's fifth joint, with a value of 20°, making the chessboard tilt upwards. On the third line of the program, a rotation of 40° is applied to the same joint, but in the opposite direction, making the chessboard go to its initial position and then tilt 20° downwards. Finally the manipulator returns to its initial position and stops its movement.

The servomotor speed used on the tilt experiments was variable (Sec. 3.6). Five trials were run with $K_p = 0.01, K_d = 0$; five with $K_p = 0.1, K_d = 0.1$; and five with $K_p = 0.12, K_d = 0.1$.

The software architecture used during the trials which used the pose estimation method to stabilize the RTU can be seen in Fig. 4.12.

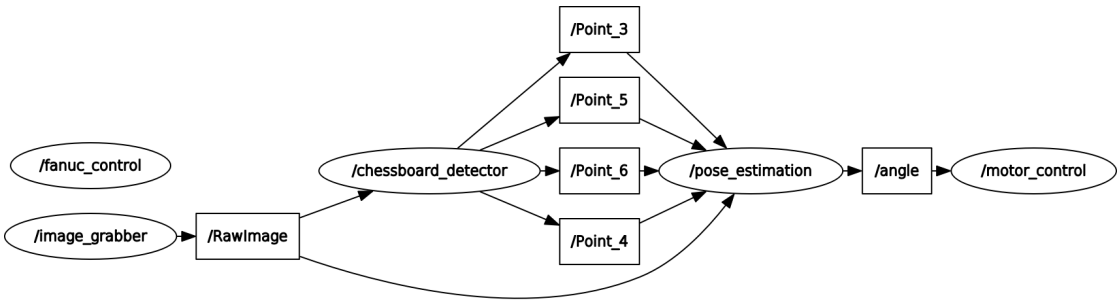


Figure 4.12: An overview of the ROS nodes and topics running during experiments on RTU stabilization using the pose estimation method.

Result analysis

To better demonstrate the results of this experiment, the plots of example trials run with the line tracking method are shown in figures 4.13, 4.14 and 4.15 and a summary of these results is present in table 4.3.

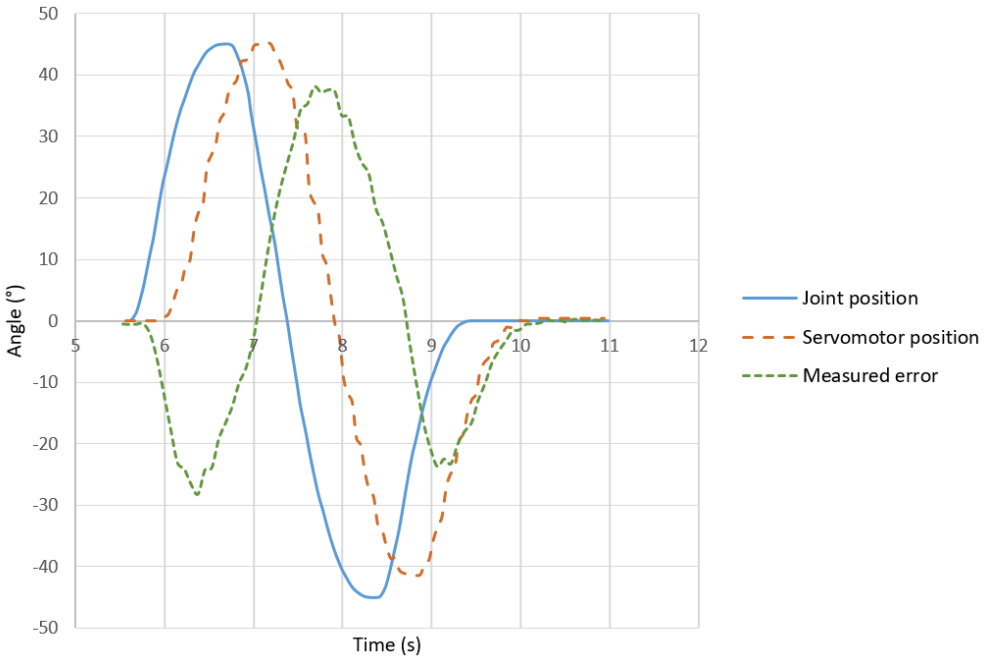


Figure 4.13: A plot of a trial run to evaluate PD control using line tracking by varying the roll angle. 15% manipulator speed. $K_p = 0.3, K_d = 0$. $\overline{E} = 13.83°$, $E_{max} = 36.68°$ and $\sigma = 9.56°$.

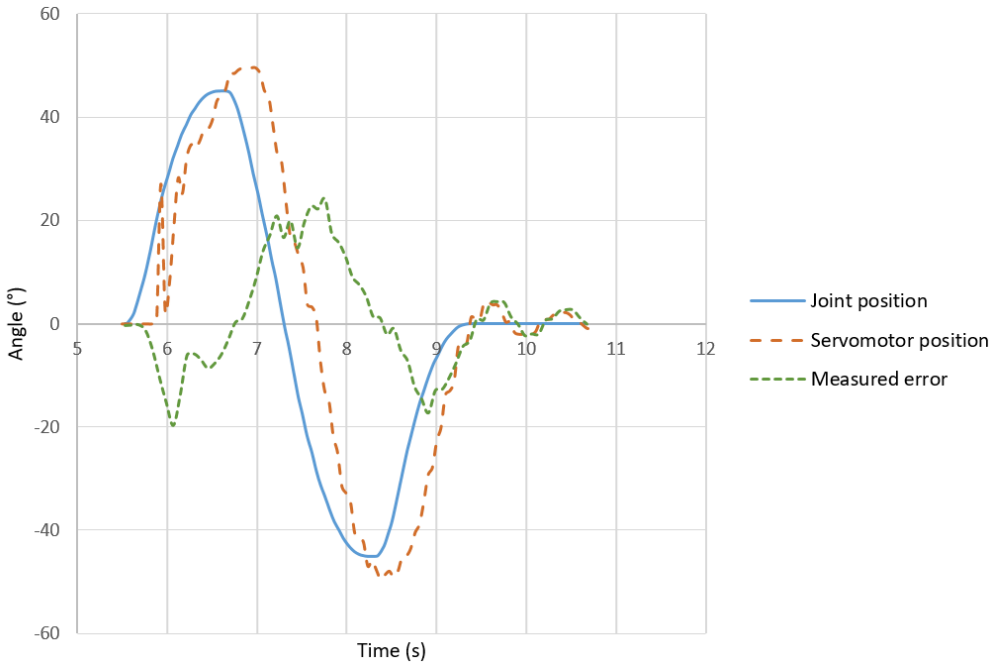


Figure 4.14: A plot of a trial run to evaluate PD control using line tracking by varying the roll angle. 15% manipulator speed. $K_p = 0.7, K_d = 0.15$. $\overline{E} = 7.79°$, $E_{max} = 24.12°$ and $\sigma = 6.90°$.

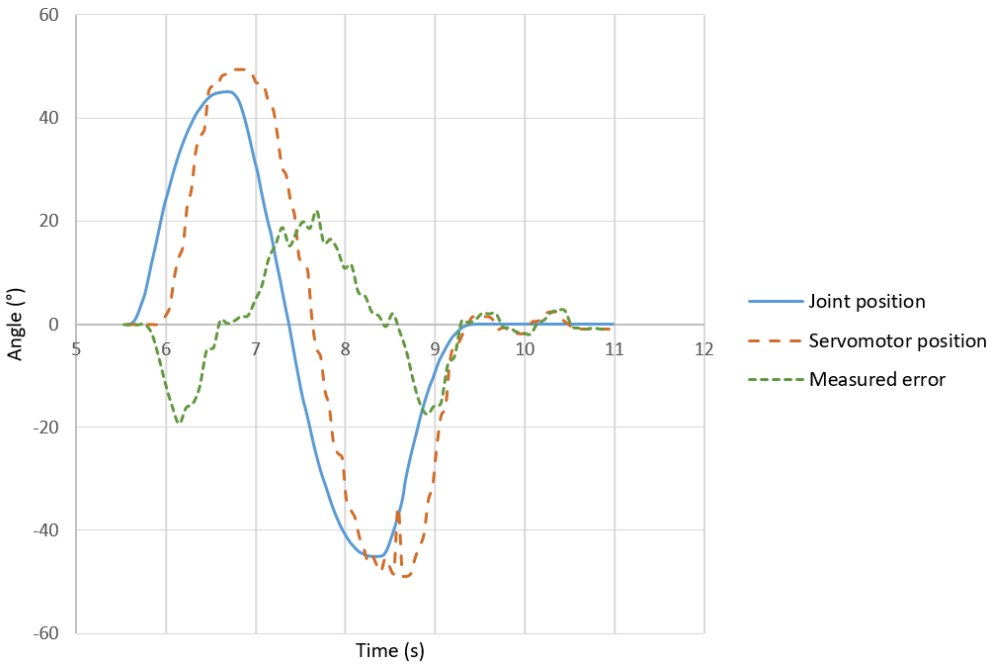


Figure 4.15: A plot of a trial run to evaluate PD control using line tracking by varying the roll angle. 15% manipulator speed. $K_p = 0.8, K_d = 0.15$. $\overline{E} = 7.68°$, $E_{max} = 21.93°$ and $\sigma = 6.93°$.

Table 4.3: PD control of RTU results. Trials run with the line tracking method.

Descriptive statistics	**Controller parameters**		
	$K_p = 0.3$	$K_p = 0.7$, $K_d = 0.15$	$K_p = 0.8$, $K_d = 0.15$
$\overline{E}$	14.19°	7.79°	7.68°
E_{max}	38.06°	24.12°	21.93°
σ	12.02°	6.90°	6.93°

The first trials started with only a small proportional coefficient and had a large error because the control algorithm was not responsive enough. After boosting the proportional component, instability and loss of the tracked chessboard was reached around the 0.8 to 0.9 values. The proportional component was assigned a final value of $K_p = 0.7$ and a derivative component was introduced with a value of $K_d = 0.15$.

Although the error still diminishes slightly when using $K_p = 0.8$ and $K_d = 0.15$, as can be seen on the plot of the example trial using these values (Fig. 4.15), the instability that this increase in the proportional component causes is too much to be considered. If these values were used, the chessboard being tracked would be lost more than half of the times that the experiment was run.

By analyzing table 4.3, it is possible to conclude that when applying a PD control with the final controller parameters, and still using the line tracking method, results were better than when using the proportional control algorithm (table 4.2).

After arriving at the final PD control parameters when using the line tracking method and varying the roll angle, the tilt angle also needed to be somehow tracked. It was at this point in the work that research was done to find the pose estimation method. Following the implementation of this method, three more types of trials were conducted for each angle to be altered. The PD control's parameters had to be re-tuned to this kind of tracking method, as it has totally different characteristics.

The plots of example trials run with the pose estimation method while varying the roll angle are shown in figures 4.16, 4.17 and 4.18 and a summary of these results is present in table 4.4.

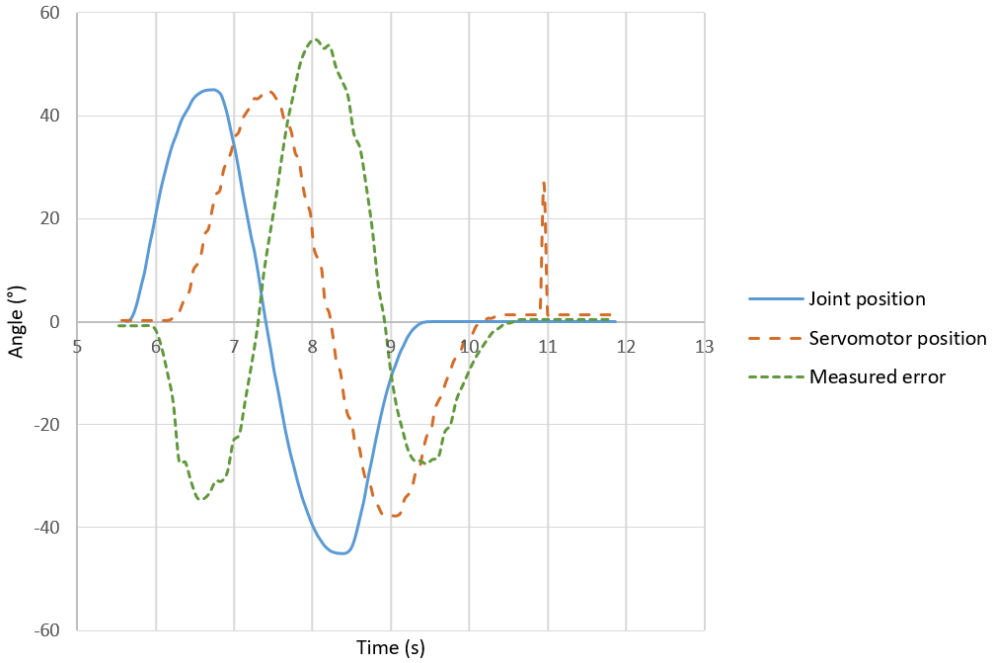


Figure 4.16: A plot of a trial run to evaluate PD control using pose estimation by varying the roll angle. 15% manipulator speed. $K_p = 0.2, K_d = 0$. $\overline{E} = 19.99°$, $E_{max} = 54.63°$ and $\sigma = 16.93°$.

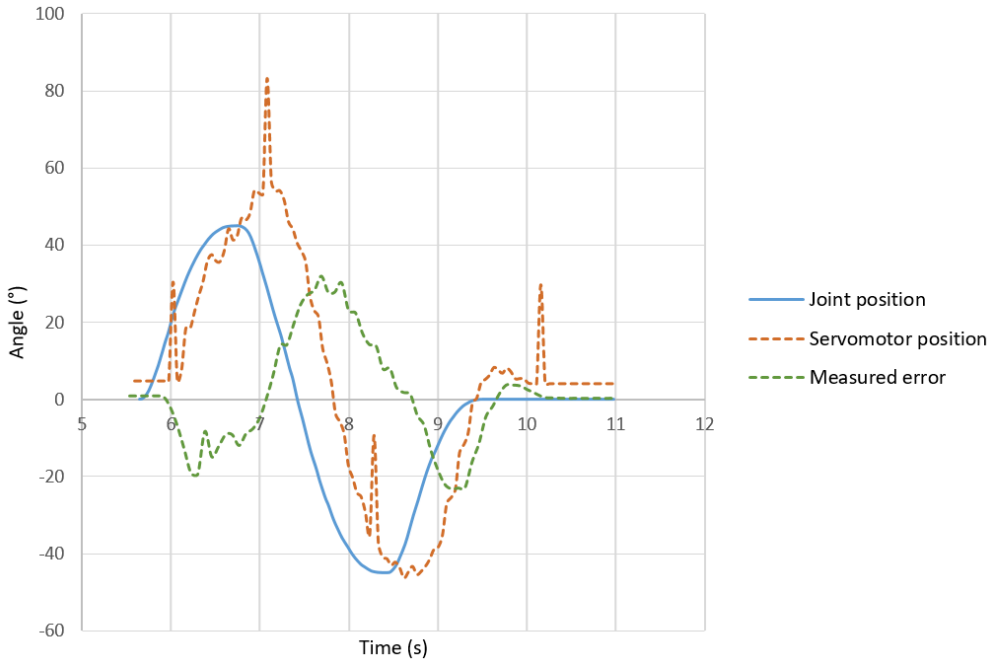


Figure 4.17: A plot of a trial run to evaluate PD control using pose estimation by varying the roll angle. 15% manipulator speed. $K_p = 0.5, K_d = 0.4$. $\overline{E} = 9.93°$, $E_{max} = 31.90°$ and $\sigma = 9.42°$.

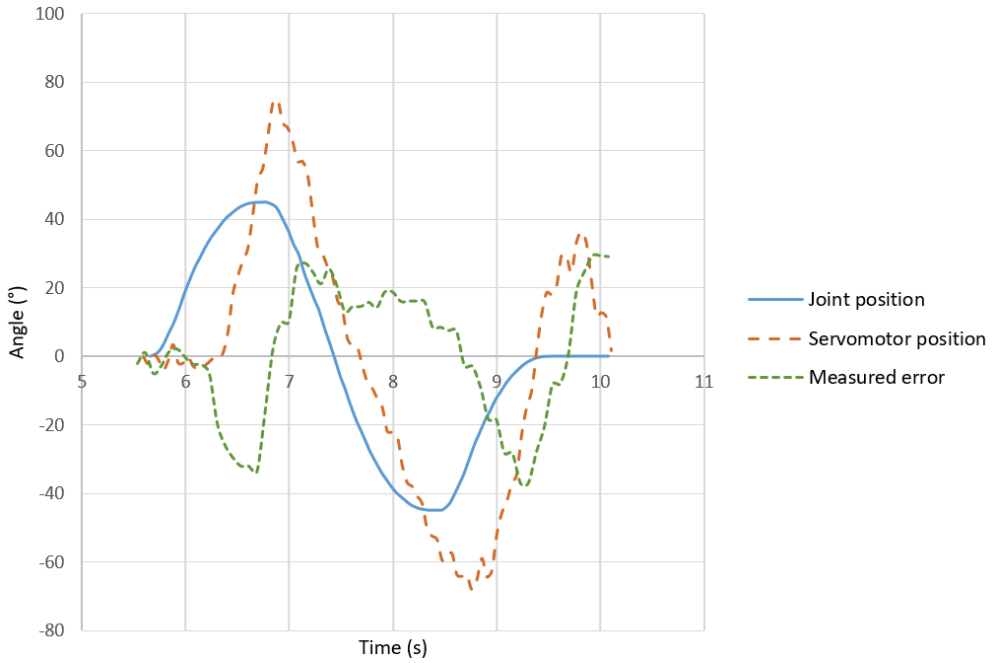


Figure 4.18: A plot of a trial run to evaluate PD control using pose estimation by varying the roll angle. 15% manipulator speed. $K_p = 0.7, K_d = 0.4$. $\overline{E} = 16.33°$, $E_{max} = 37.18°$ and $\sigma = 10.51°$.

Table 4.4: PD control of RTU results. Trials run with the pose estimation method while varying the roll angle.

Descriptive statistics	**Controller parameters**		
	$K_p = 0.2$	$K_p = 0.5,\ K_d = 0.4$	$K_p = 0.7,\ K_d = 0.4$
$\overline{E}$	19.99°	9.93°	16.33°
E_{max}	54.63°	31.90°	37.18°
σ	16.93°	9.42°	10.51°

The first trial run whilst using the pose estimation method to measure a change in the roll angle was made with $K_p = 0.2$ and no derivative component, following the same train of thought as the previous experiments. Once again, the control algorithm is not responsive enough and as such, the error values are rather large. After extensive testing, the final values reached for the PD control's parameters when using the pose estimation method were $K_p = 0.5$ and $K_d = 0.4$. A plot of an example trial with these values can be seen in Fig. 4.17.

The last trials varying the roll angle are meant to show that when using values greater than the ones that were established, besides not yielding far better results, when the RTU was reaching its final position it could become unstable, balancing back and forth without ever stabilizing. An example of a trial of this type is shown in Fig. 4.18.

By analyzing table 4.4 we can see that the best results are obtained with the final values of $K_p = 0.5$ and $K_d = 0.4$. However, these are worse than the results shown in table 4.3, which were acquired using the line tracking method. This is thought to be

because of the update rate of the pose estimation method being visibly lower than the one of the line tracking method.

The plots of example trials run with the pose estimation method while varying the tilt angle are shown in figures 4.19, 4.20 and 4.21 and a summary of these results is present in table 4.5.

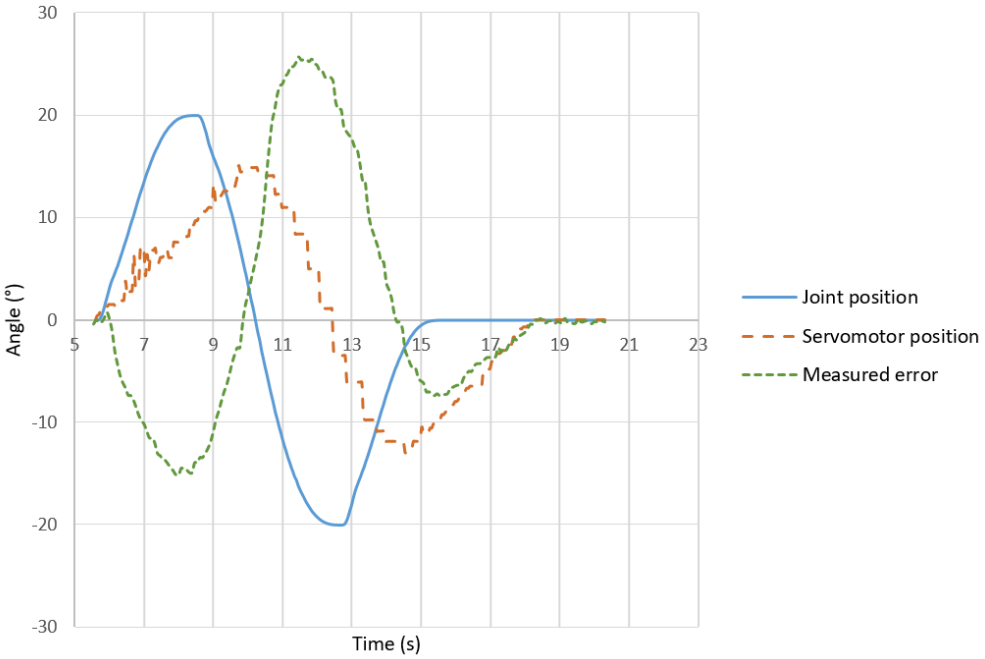


Figure 4.19: A plot of a trial run to evaluate PD control using pose estimation by varying the tilt angle. 5% manipulator speed. $K_p = 0.01, K_d = 0$. $\overline{E} = 8.62°$, $E_{max} = 25.72°$ and $\sigma = 7.84°$.

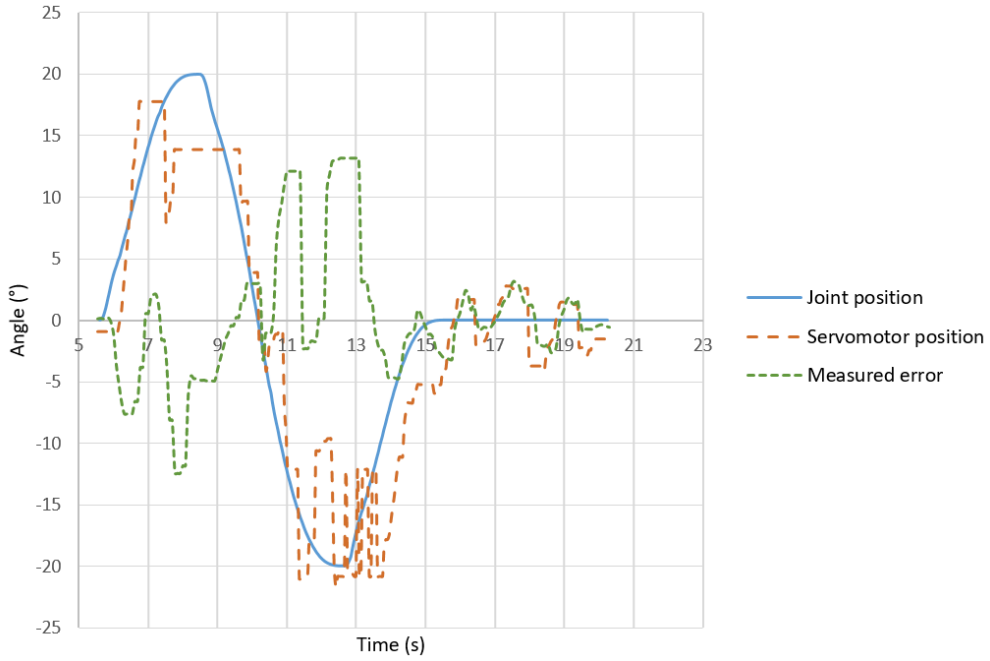


Figure 4.20: A plot of a trial run to evaluate PD control using pose estimation by varying the tilt angle. 5% manipulator speed. $K_p = 0.1, K_d = 0.1$. $\overline{E} = 3.51°$, $E_{max} = 13.13°$ and $\sigma = 3.87°$.

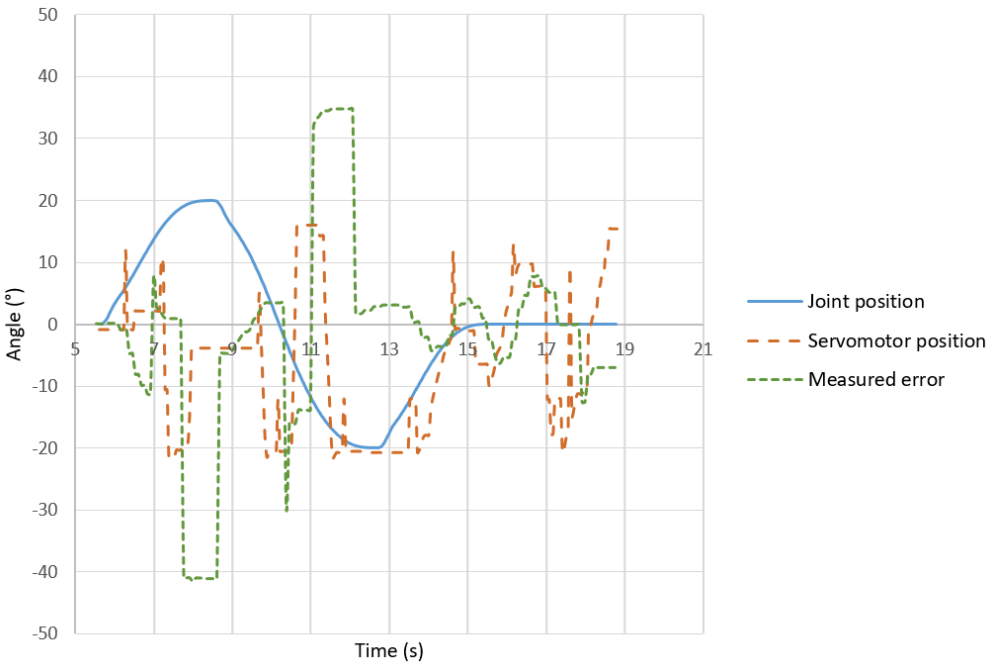


Figure 4.21: A plot of a trial run to evaluate PD control using pose estimation by varying the tilt angle. 5% manipulator speed. $K_p = 0.12, K_d = 0.1$. $\overline{E} = 9.33°$, $E_{max} = 41.60°$ and $\sigma = 12.59°$.

Table 4.5: PD control of RTU results. Trials run with the pose estimation method while varying the tilt angle.

Descriptive statistics	**Controller parameters**		
	$K_p = 0.01$	$K_p = 0.1$, $K_d = 0.1$	$K_p = 0.12$, $K_d = 0.1$
$\overline{E}$	8.62°	3.51°	9.33°
E_{max}	25.72°	13.13°	41.60°
σ	7.84°	3.87°	12.59°

Although the tilt servo uses a different kind of PD control (velocity control)(sec. 3.6), the procedure for its tuning is the same. The first component to introduce was the proportional component with a $K_p = 0.01$ and to demonstrate that the control algorithm is not responsive enough the results of an example trial can be seen in Fig. 4.19.

The final values reached for the PD control when varying the tilt angle were $K_p = 0.1$ and $K_d = 0.1$. A plot of an example trial with these values can be seen in Fig. 4.20. This plot also shows the problem that was the origin of having to use a different kind of control for the servo responsible for the tilt angle. As is seen in the servomotor position line, it is impossible to do a PD control where the control variable is the servo's position if the current position of the servo is retrieved so badly. This mostly happens with the tilt servomotor for reasons explained in section 3.6.

From $K_p = 0.12$ and onwards, high instability occurred when the RTU was reaching its final position and once again, the servo would go back and forth, without ever stabilizing.

By analyzing table 4.5, it is possible to see the progressive tuning of the tilt angle servo's speed PD controller's parameters. The best results are obtained with $K_p = 0.1$ and $K_d = 0.1$ and values higher than this resulted in an instability position of the RTU at the end of experiments.

4.3 Experiments with RTU mounted on the manipulator arm

4.3.1 Experiments description

After all of the experiments where the chessboard was mounted on the manipulator arm were done, the RTU was mounted on the manipulator (Fig. 4.22) to see if the response to the manipulator's movements was similar. To mount the RTU on the manipulator, some of the previous mechanical parts designed were used, with the only extra alteration being two threaded holes on the RTU support.

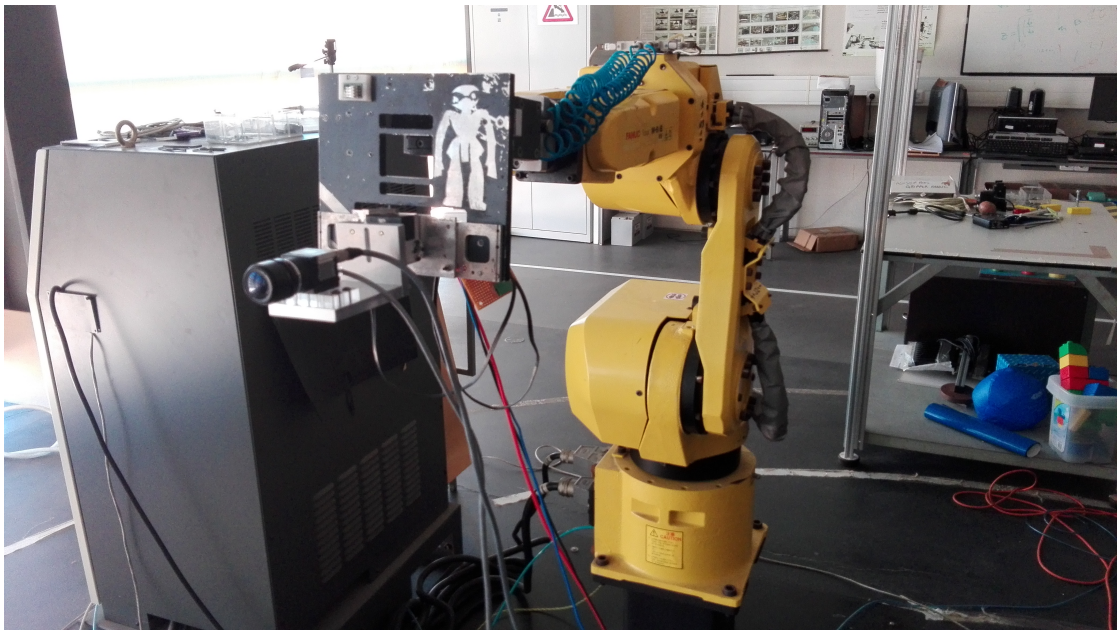


Figure 4.22: The RTU mounted on the manipulator arm.

In these experiments, the control method used was solely the PD control method but both tracking methods were tested. To do so, five trials were run with the line tracking method where $K_p = 0.6, K_d = 0.15$; five trials with the pose estimation method for the roll angle where $K_p = 0.6, K_d = 0.4$; and five trials with the pose estimation method for the tilt angle where $K_p = 0.08, K_d = 0.02$.

The distance to the chessboard was still two meters for all trials and servomotor speed was 255 for experiments where the roll angle was being tested. Additionally, two TP programs were created, called `JS_ROLL2` and `JS_TILT2`. These are the same as the TP programs explained above, only with an offset on the fifth axis of the manipulator in order to start the experiments with the RTU leveled with the ground.

4.3.2 Result analysis

The plots of example trials run with the RTU mounted in the manipulator are shown in figures 4.23, 4.24 and 4.25 and a summary of these results is present in table 4.6.

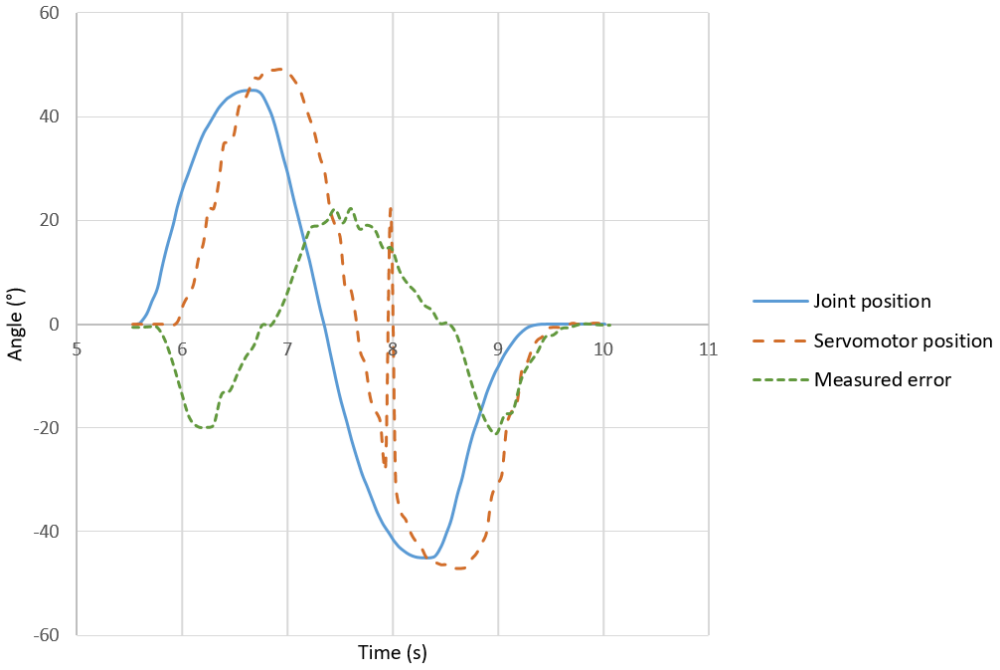


Figure 4.23: A plot of a trial run to evaluate PD control using line tracking by varying the roll angle. 15% manipulator speed. $K_p = 0.6, K_d = 0.15. \overline{E} = 7.54°$, $E_{max} = 22.35°$ and $\sigma = 7.81°$.

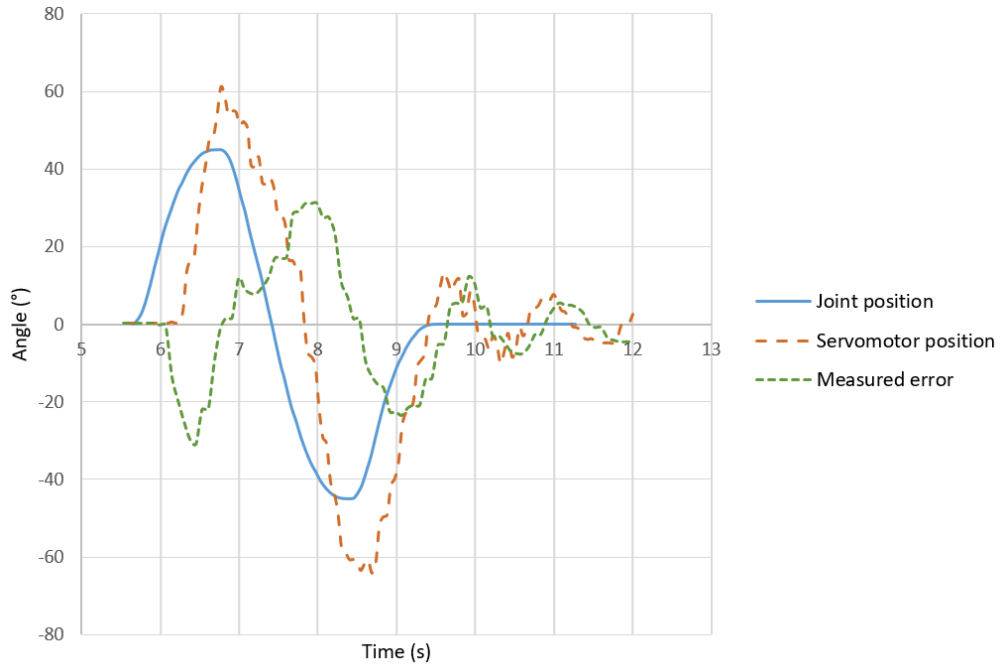


Figure 4.24: A plot of a trial run to evaluate PD control using pose estimation by varying the roll angle. 15% manipulator speed. $K_p = 0.6, K_d = 0.4$. $\overline{E} = 13.64°$, $E_{max} = 31.05°$ and $\sigma = 9.96°$.

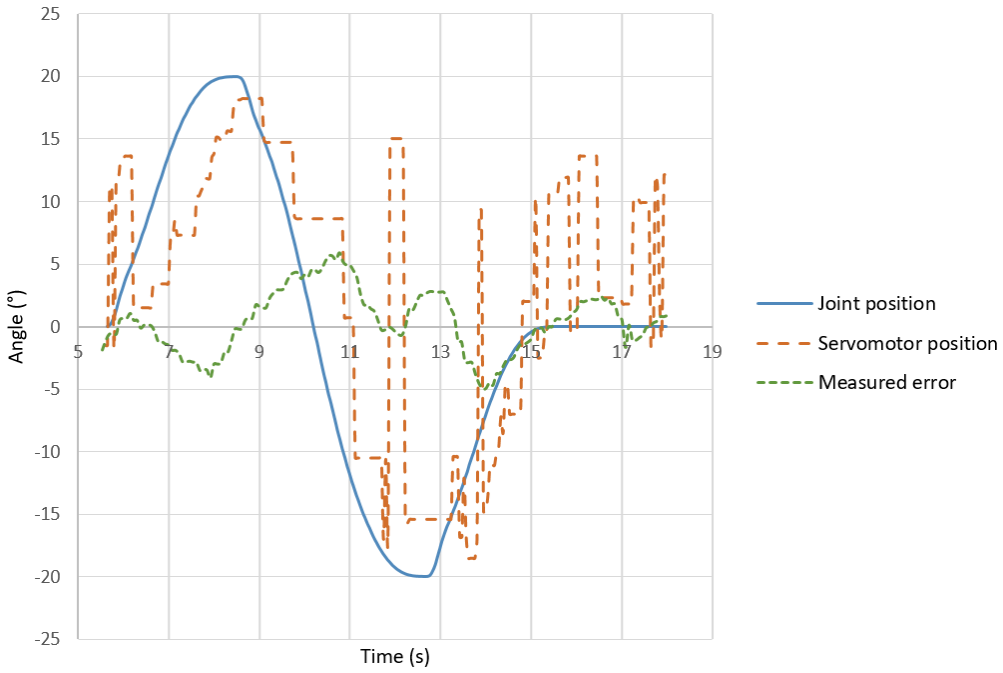


Figure 4.25: A plot of a trial run to evaluate PD control using pose estimation by varying the tilt angle. 5% manipulator speed. $K_p = 0.08, K_d = 0.02$. $\overline{E} = 2.01°$, $E_{max} = 5.88°$ and $\sigma = 1.52°$.

Table 4.6: Results of experiments performed with the RTU mounted on the manipulator.

	Descriptive statistics	**Controller parameters** $K_p = 0.6$, $K_d = 0.15$
Line tracking - Roll	$\overline{E}$	7.54°
	E_{max}	22.35°
	σ	7.81°
		Controller parameters $K_p = 0.6$, $K_d = 0.4$
Pose estimation - Roll	$\overline{E}$	13.64°
	E_{max}	31.05°
	σ	9.96°
		Controller parameters $K_p = 0.08$, $K_d = 0.02$
Pose estimation - Tilt	$\overline{E}$	2.01°
	E_{max}	5.88°
	σ	1.52°

A plot of an example trial where the line tracking method was used can be seen in Fig. 4.23. The PD control's parameters had to be slightly adjusted to avoid instability

and the final values were $K_p = 0.6$ and $K_d = 0.15$.

Regarding the pose estimation trials, the PD control's parameters were also changed when compared to the previous values where the chessboard was mounted on the manipulator. When varying the roll angle, the values used were $K_p = 0.6$ and $K_d = 0.4$ and a plot of an example trial run with those values can be seen in Fig. 4.24.

When running the tilt trials, the values $K_p = 0.08$ and $K_d = 0.02$ were used. A plot of an example trial run with those values can be seen in Fig. 4.25.

Overall, these results proved to be similar to what happened when the chessboard was mounted on the manipulator and the RTU was mounted on the tripod, and confirm that no vibrations or disturbances coming from the manipulator interfere with the tracking or stabilization processes.

By analyzing table 4.6, it is possible to see that although needing a small alteration in the PD values, the results are very similar to what happened in the experiments where the chessboard was mounted on the manipulator, with a slight increase in the mean error in the pose estimation - roll experiment.

Chapter 5

Conclusions and future work

In this chapter, the main conclusions that have come from the work done in this dissertation will be presented, along with proposals of future work in the area that can be integrated into PHUA.

5.1 Conclusions

Regarding the accomplishment of this dissertation's objectives, it can be said that only some of them were accomplished. When it comes to estimating the direction of the gravity vector using both visual and inertial data, the first main objective of this dissertation, more focus was put into using visual data, as was planned. However, to test if the inertial sensors were still working properly in case this is to be addressed in a future work, the Arduino board was connected to the computer and the relevant ROS nodes developed previously in [5] were compiled. The Arduino code which would read the sensor's values was also uploaded to the board and it was possible to visualize the inertial readings in real-time on the computer screen.

Both the roll and tilt components of the gravity vector were estimated successfully, by using two methods based on tracking an object present in the image frame.

The first method used (referred to as the line tracking method), tracks a line formed by the squares of a chessboard who was placed in front of the camera mounted in the RTU (humanoid head). This method is able to provide the angle of this line, which in turn allows the estimation of the roll component of the gravity vector, provided that the orientation of the chessboard in relation to the environment is known.

The second method used (referred to as the pose estimation method), estimates the pose (position and orientation) of the chessboard mentioned in the previous paragraph in relation to the RTU, which provides both the roll and tilt components of the gravity vector, if the pose of the chessboard relative to the environment is known.

The second objective of this dissertation, which was to develop computational tools which would act on the RTU, based on the previously estimated verticality and allowing it to remain stable, was also fulfilled. The final solution involved a PD controller whose control variable was the position of the servo responsible for the roll angle of the RTU and a PD controller whose control variable was the speed of the servo responsible for the tilt angle of the RTU. Although stabilization for both the roll and tilt components of the RTU could not be performed at the same time, it was successfully done separately. This

is because the existing infrastructure was not suitable for this purpose (more details in section 3.6 - Fig. 3.8), as it was thought to be in the beginning of this work.

Concerning the third and final objective of the dissertation, which was to explore and demonstrate the benefits of head stabilization when trying to achieve a humanoid robot's balance, it can be said that it was not completely fulfilled. Although the benefits of head stabilization were explored, they could not be demonstrated since the PHUA platform was not functional at the time of this work. The humanoid robot was undergoing repairs by volunteers at LAR, but these were not finished in time to allow the demonstration of the benefits of head stabilization.

The experiments carried out during the dissertation helped show the progressive tuning of each of the PD controller's parameters and allowed for the evaluation of the line tracking method. Thinking back, if a change was to be made to the experiments, it would be the addition of a new experiment to evaluate the pose estimation method. Since this method was implemented after experiments regarding the RTU's stabilization had begun, the importance of such an experiment was overlooked. As a final remark to the experiment's results, it was also possible to conclude that when stabilization was done using the line tracking method to estimate the gravity vector's roll component, results were slightly better than when using the pose estimation method. This was thought to be due to the decreased update rate of the pose estimation method when compared to the line tracking method.

One of the biggest challenges surpassed during this dissertation was to gather and apply all the knowledge and methods used in previous works, which dated back to the creation of LAR and the PHUA project itself. Another topic that should be mentioned is the possible inadequacy of the camera's frame rate to perform tracking operations. Although the camera used has a great resolution, a maximum frame rate of 15 frames per second is too low to perform real-time stabilization of a humanoid robot. The servomotors used also did not perform fully as expected, as one of them showed problems when returning its current angular position when requested.

As a final note, it is important to say that the FANUC robotic manipulator used proved invaluable when providing a ground truth for the experiments that were carried out.

5.2 Future work

This dissertation allowed the development of the work started in previous theses on a vision-based balance system. However, there are many more tasks that need to be done in order to integrate this new balance system into the PHUA project, and some are listed below:

- Analyze if the pose estimation tool of the ViSP library can be used more effectively (i.e. increase its update rate);
- Combine the estimation of the gravity vector using visual data with an estimation using inertial data when it is not plausible to use visual data (e.g. poor lighting, movements that are too sudden, etc.)
- Further tune the PD controller's parameters used to stabilize the humanoid head.

- Obtain a camera which is more suitable for stabilization purposes (i.e. higher frame rate) and install it in the humanoid head.
- Verify if it is viable to install a mini-computer on the PHUA robot (e.g Raspberry Pi), which would perform all calculations regarding its balance, and other operations, thus making the robot more autonomous.
- Demonstrate the benefits of head stabilization when trying to achieve a humanoid robot's balance.

References

[1] Shuuji Kajita et al. *Introduction to Humanoid Robotics*. English. Vol. 101. Springer Tracts in Advanced Robotics. Springer, 2014.

[2] *DARPA Robotics Challenge (DRC)*. URL: `http://www.darpa.mil/program/darpa-robotics-challenge` (visited on 05/26/2017).

[3] *humanoids.jpg (JPEG Image, 470 × 358 pixels)*. URL: `https://s-media-cache-ak0.pinimg.com/736x/13/dd/81/13dd815712f010b1bc20cf2a92bda43e.jpg` (visited on 05/31/2017).

[4] Vítor Santos, Rui Moreira, and Filipe Silva. "Mechatronic Design of a New Humanoid Robot with Hybrid Parallel Actuation". In: *International Journal of Advanced Robotic Systems* 9.119 (2012).

[5] Telmo Rafeiro. "Rede de Sensores Inerciais para Equilíbrio de um Robô Humanóide". Master's Thesis. Universidade de Aveiro, 2013.

[6] João Barros. "Cooperative haptics for humanoid robot teleoperation". Master's Thesis. Universidade de Aveiro, 2014.

[7] João Peixoto. "Visual and Inertial Data Integration to Assist Humanoid Balance". Master's Thesis. Universidade de Aveiro, 2016.

[8] César Sousa. "Visual Servoing of a Humanoid Head Using Fixed Targets". Master's Thesis. Universidade de Aveiro, 2016.

[9] João Barros, Vítor Santos, and Filipe Silva. "Tele-Kinesthetic Teaching of Motion Skills to Humanoid Robots through Haptic Feedback". In: IEEE-RAS Humanoids 2014 Workshop on Policy Representations for Humanoid Robots.

[10] Emílio Estrelinha. "Tele-operation of a humanoid robot using Haptics and Load Sensors". Master's Thesis. Universidade de Aveiro, 2013.

[11] E. Marchand, F. Spindler, and F. Chaumette. "ViSP for visual servoing: a generic software platform with a wide class of robot control skills". In: *IEEE Robotics and Automation Magazine* 12.4 (Dec. 2005), pp. 40–52.

[12] Valerie C. Scanlon and Tina Sanders. *Essentials of Anatomy and Physiology*. Fifth edition. F. A. Davis Company, 2007.

[13] *The Human Balance System | Vestibular Disorders Association*. URL: `http://vestibular.org/understanding-vestibular-disorder/human-balance-system` (visited on 11/06/2017).

[14] Miomir Vukobratović and J. Stepanenko. “On The Stability of Anthropomorphic Systems”. In: *Mathematical Biosciences* 15.1-2 (1972), pp. 1–37.

[15] Alain Berthoz. “On the Benefits of Head Stabilization with a View to Control Balance and Locomotion in Humanoids”. In: 11th IEEE-RAS International Conference on Humanoid Robots, 2011, pp. 147–152.

[16] *Flea3 2.8 MP Color GigE Vision (Sony ICX687)*. URL: `https://www.ptgrey.com/flea3-28-mp-color-gige-vision-sony-icx687-camera` (visited on 06/03/2017).

[17] *HSR-5498SG HMI Premium Robot Servo | HITEC RCD USA*. URL: `http://hitecrcd.com/products/servos/discontinued-servos-servo-accessories/hsr-5498sg-hmi-premium-robot-servo/product` (visited on 06/03/2017).

[18] Lee Jun Hee. *General Specification of HSR-5498SG Digital Robot Servo*. June 2016.

[19] *Arduino - Introduction*. URL: `https://www.arduino.cc/en/Guide/Introduction` (visited on 06/06/2017).

[20] *Arduino - Reference*. URL: `https://www.arduino.cc/en/Reference/HomePage` (visited on 06/06/2017).

[21] *Arduino - Software*. URL: `https://www.arduino.cc/en/Main/Software` (visited on 06/06/2017).

[22] *Arduino - ArduinoBoardUno*. URL: `https://www.arduino.cc/en/Main/ArduinoBoardUno` (visited on 06/06/2017).

[23] *9 Degrees of Freedom - Razor IMU - SEN-10736 - SparkFun Electronics*. URL: `https://www.sparkfun.com/products/retired/10736` (visited on 06/09/2017).

[24] InvenSense. *ITG-3200 Product Specification Revision 1.4*. Mar. 2010.

[25] Analog Devices. *ADXL345, 3-axis Digital Accelerometer*. 2009.

[26] Honeywell. *3-axis Digital Compass IC HMC5883L*. Mar. 2011.

[27] *Pololu - MinIMU-9 v2 Gyro, Accelerometer, and Compass (L3GD20 and LSM303DLHC Carrier)*. URL: `https://www.pololu.com/product/1268` (visited on 06/09/2017).

[28] ST. *L3GD20, MEMS motion sensor: three-axis digital output gyroscope, datasheet - production data*. Feb. 2013.

[29] ST. *LSM303DLHC, Ultra compact high performance e-compass 3D accelerometer and 3D magnetometer module*. Feb. 2013.

[30] FANUC Robotics. *M-6iB Series*. June 2007.

[31] *FANUC M-6iB/6S RJ3iB*. URL: `https://www.robots.com/fanuc/m-6ib-6s` (visited on 06/12/2017).

[32] *Ubuntu PC operating system | Ubuntu*. URL: `https://www.ubuntu.com/desktop` (visited on 09/30/2017).

[33] *ROS.org | About ROS*. URL: `http://www.ros.org/about-ros/` (visited on 09/30/2017).

[34] *ROS/Introduction - ROS Wiki*. URL: http://wiki.ros.org/ROS/Introduction (visited on 09/30/2017).

[35] *ROS.org | Is ROS For Me?* URL: http://www.ros.org/is-ros-for-me/ (visited on 09/30/2017).

[36] *Exchange Data with ROS Publishers and Subscribers - MATLAB & Simulink*. URL: https://www.mathworks.com/help/robotics/examples/exchange-data-with-ros-publishers-and-subscribers.html (visited on 09/30/2017).

[37] *About - OpenCV library*. URL: https://opencv.org/about.html (visited on 10/13/2017).

[38] *Introduction — OpenCV 2.4.13.4 documentation*. URL: https://docs.opencv.org/2.4/modules/core/doc/intro.html (visited on 10/13/2017).

[39] *ViSP*. URL: https://visp.inria.fr/ (visited on 09/30/2017).

[40] *Software architecture – ViSP*. URL: https://visp.inria.fr/software-architecture/ (visited on 10/01/2017).

[41] *Moving-edges – ViSP*. URL: http://visp.inria.fr/moving-edges/ (visited on 10/01/2017).

[42] *Computer vision – ViSP*. URL: http://visp.inria.fr/computer-vision/ (visited on 10/01/2017).

[43] *Visual Servoing Platform: Tutorial: Pose estimation from points*. URL: http://visp-doc.inria.fr/doxygen/visp-daily/tutorial-pose-estimation.html (visited on 10/01/2017).

[44] *Arduino - Servo*. URL: https://www.arduino.cc/en/reference/servo (visited on 10/01/2017).

[45] Rui Cancela. "Extensão e flexibilização da interface de controlo de um manipulador robótico FANUC". Master's Thesis. Universidade de Aveiro, 2007.

[46] Rui Cancela. "Extensão e flexibilização da interface de controlo de um manipulador robótico FANUC". Anexo C. Lista de Funções robCOMM. Master's Thesis. Universidade de Aveiro, 2007.

[47] *Glade - A User Interface Designer*. URL: https://glade.gnome.org/ (visited on 10/01/2017).

[48] A. Carolina Matos. "Development of a large baseline stereo vision rig for pedestrian and other target detection on road". Master's Thesis. Universidade de Aveiro, 2016.

[49] David Silva. "Multisensor Calibration and Data Fusion Using LIDAR and Vision". Master's Thesis. Universidade de Aveiro, 2016.

[50] FLIR. *FLIR FLEA 3 GigE Vision - Technical Reference*. Jan. 2017.

[51] *cv_bridge - ROS Wiki*. URL: http://wiki.ros.org/cv_bridge (visited on 10/13/2017).

[52] *camera_calibration/Tutorials/MonocularCalibration - ROS Wiki*. URL: http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration (visited on 10/15/2017).

[53] Pedro Cruz. “Haptic interface data acquisition system”. Master’s Thesis. Universidade de Aveiro, 2012.

[54] *LAR toolkit v4*. URL: `http://lars.mec.ua.pt/lartk4/` (visited on 10/16/2017).

[55] *rosbag - ROS Wiki*. URL: `http://wiki.ros.org/rosbag` (visited on 10/28/2017).

[56] Nick Speal. *ROStools: This repo is a home for tools I’ve built to share with the ROS community*. Nov. 2014. URL: `https://github.com/nickspeal/ROStools` (visited on 10/28/2017).

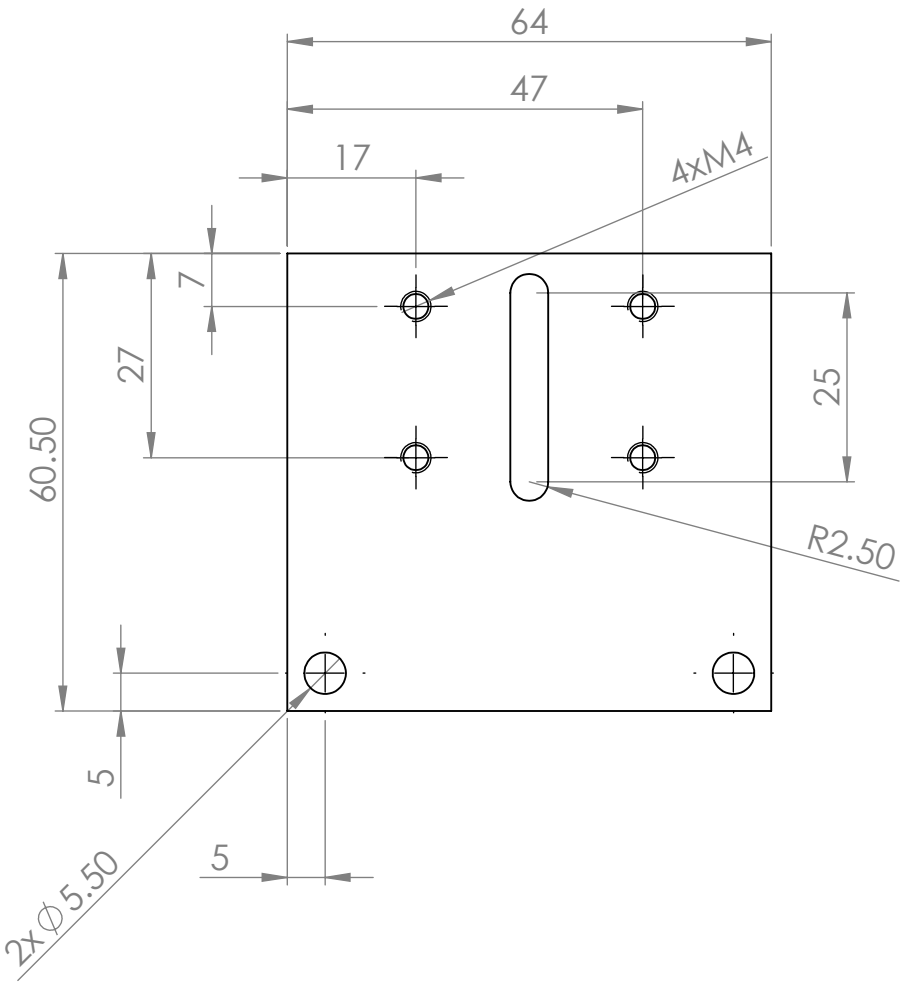
[57] *roslaunch - ROS Wiki*. URL: `http://wiki.ros.org/roslaunch` (visited on 10/28/2017).

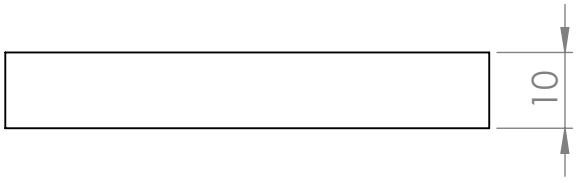
[58] *rqt_graph - ROS Wiki*. URL: `http://wiki.ros.org/rqt_graph` (visited on 10/28/2017).

Appendix A

Technical Drawings

PTU to RTU transformation





UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:	DEBURR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING / REVISION

	NAME	SIGNATURE	DATE
DRAWN			
CHK'D			
APPV'D			
MFG			
Q.A			

TITLE:

PTU to RTU transformation - part 1

MATERIAL:

DWG NO.

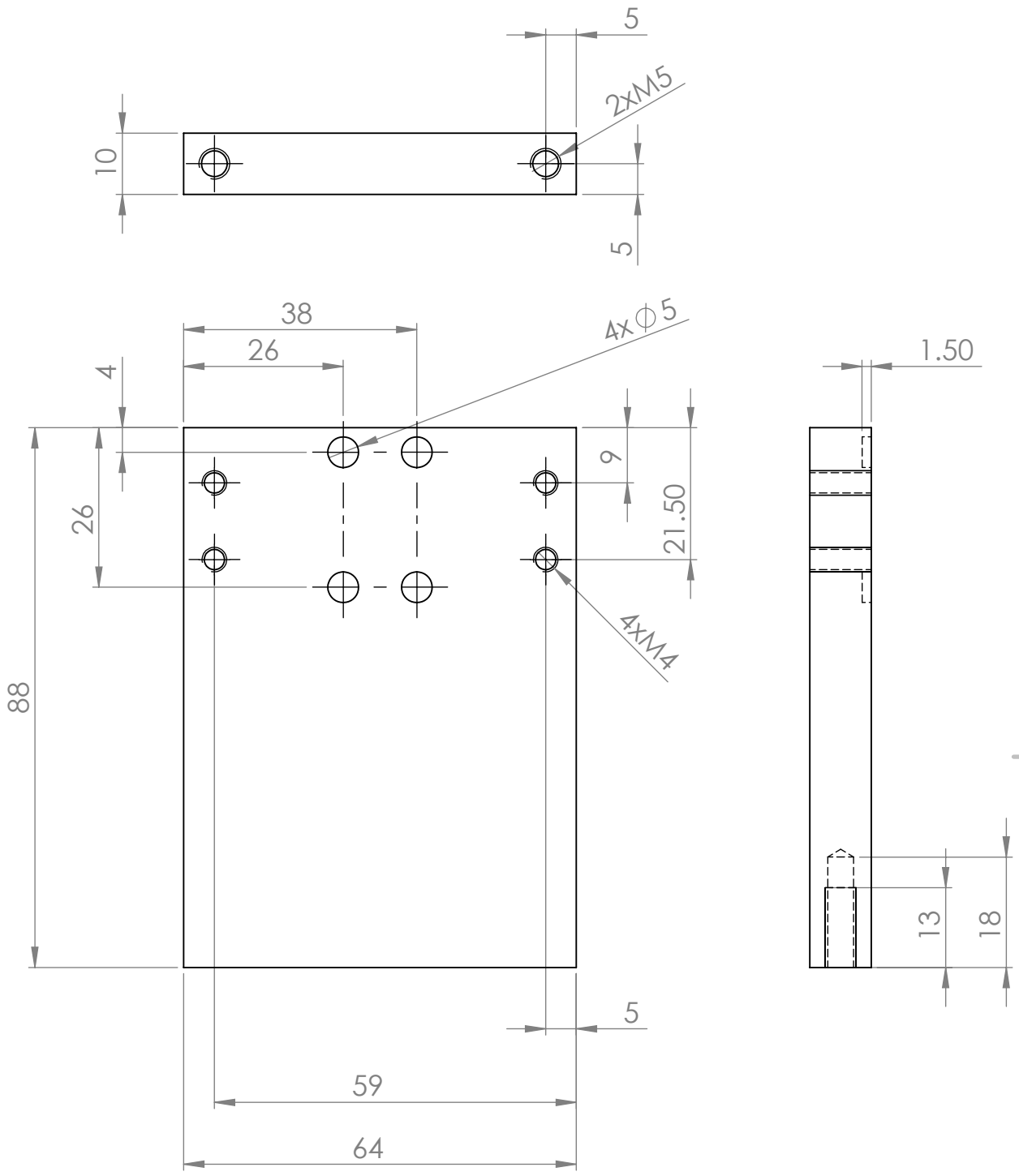
A4

WEIGHT:

SCALE:1:1

SHEET 1 OF 1

SOLIDWORKS Educational Product. For Instructional Use Only.



UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:	DEBURR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING / REVISION

	NAME	SIGNATURE	DATE
DRAWN			
CHK'D			
APPV'D			
MFG			
Q.A			

TITLE:

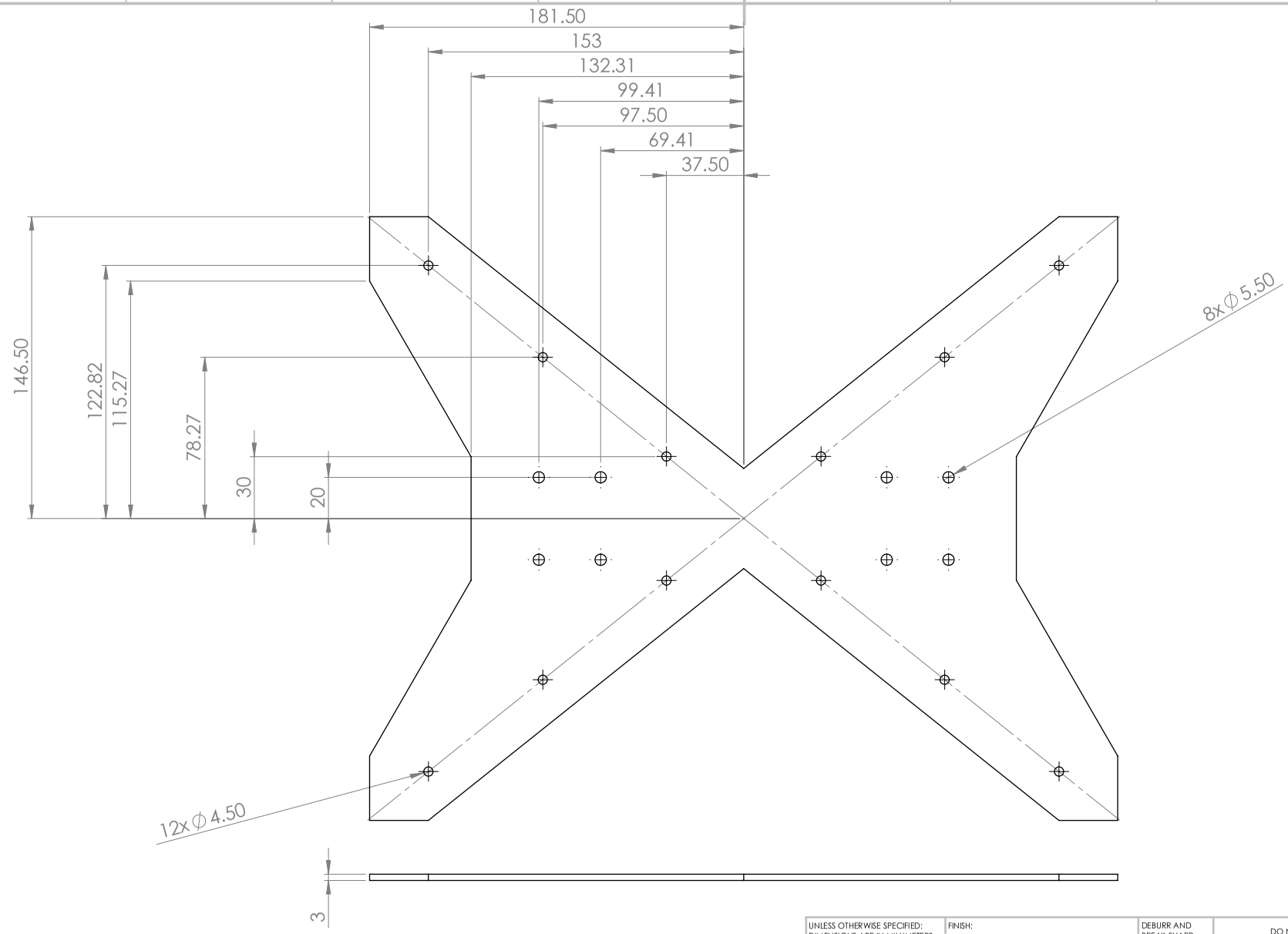
PTU to RTU transformation - part 2

MATERIAL:	DWG NO.	A4
WEIGHT:	SCALE:1:1	SHEET 1 OF 1

SOLIDWORKS Educational Product. For Instructional Use Only.

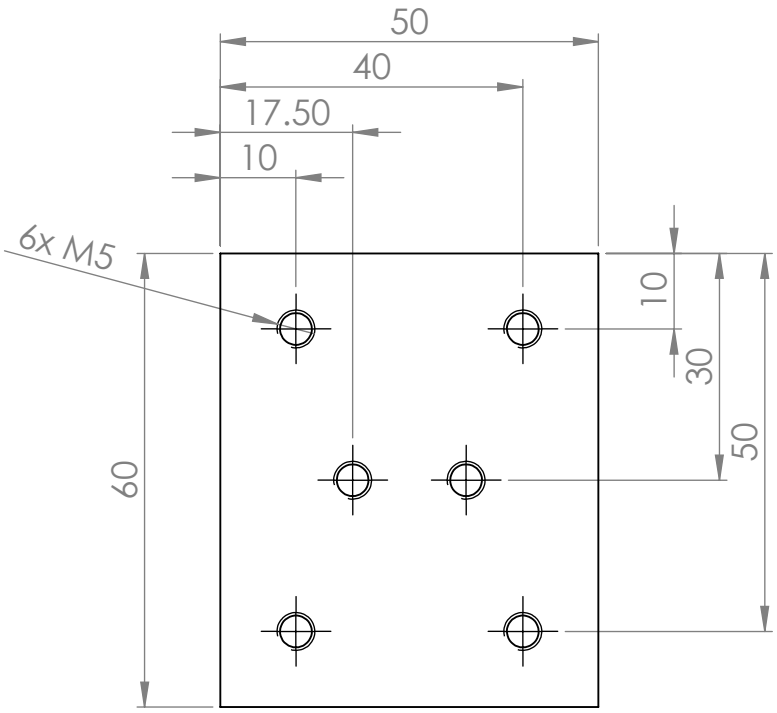
Chessboard connection to manipulator

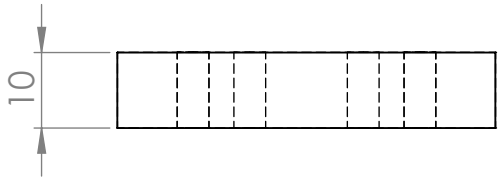
Note: The "Chessboard back - part 1" drawing was rescaled to fit onto an A4 page.



UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:		DEBURR AND BREAK SHARP EDGES		DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE			TITLE:	
DRAWN						Chessboard back - part 1	
CHK'D							
APPV'D							
MFG							
Q.A				MATERIAL:		DWG NO.	A3
				WEIGHT:		SCALE:1:2	SHEET 1 OF 1

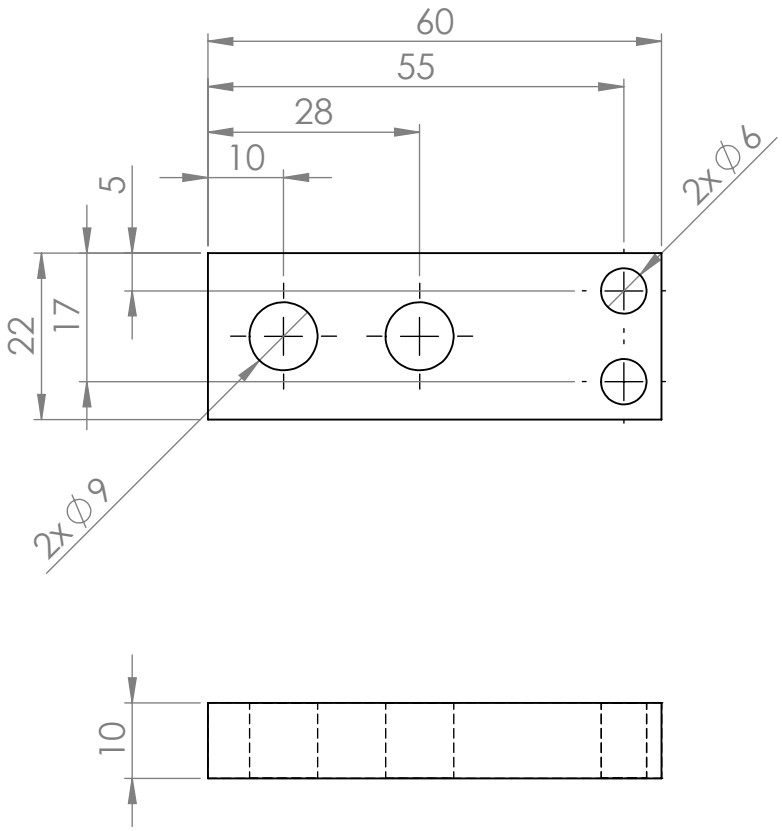
SOLIDWORKS Educational Product. For Instructional Use Only.





UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBURR AND BREAK SHARP EDGES			DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE			TITLE: Chessboard back - part two	
DRAWN							
CHK'D							
APPV'D							
MFG							
Q.A				MATERIAL:		DWG NO.	A4
				WEIGHT:		SCALE:1:1	SHEET 1 OF 1

SOLIDWORKS Educational Product. For Instructional Use Only.



UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:	DEBURR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING / REVISION

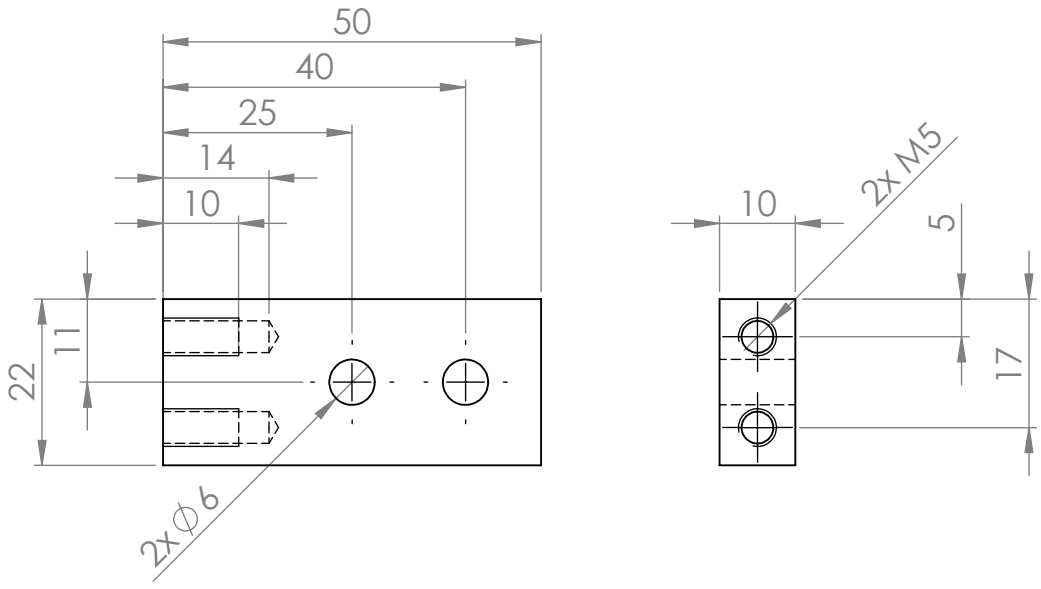
	NAME	SIGNATURE	DATE
DRAWN			
CHK'D			
APPV'D			
MFG			
Q.A			

TITLE:

Chessboard connection to manipulator - part 1

MATERIAL:	DWG NO.	A4
WEIGHT:	SCALE:1:1	SHEET 1 OF 1

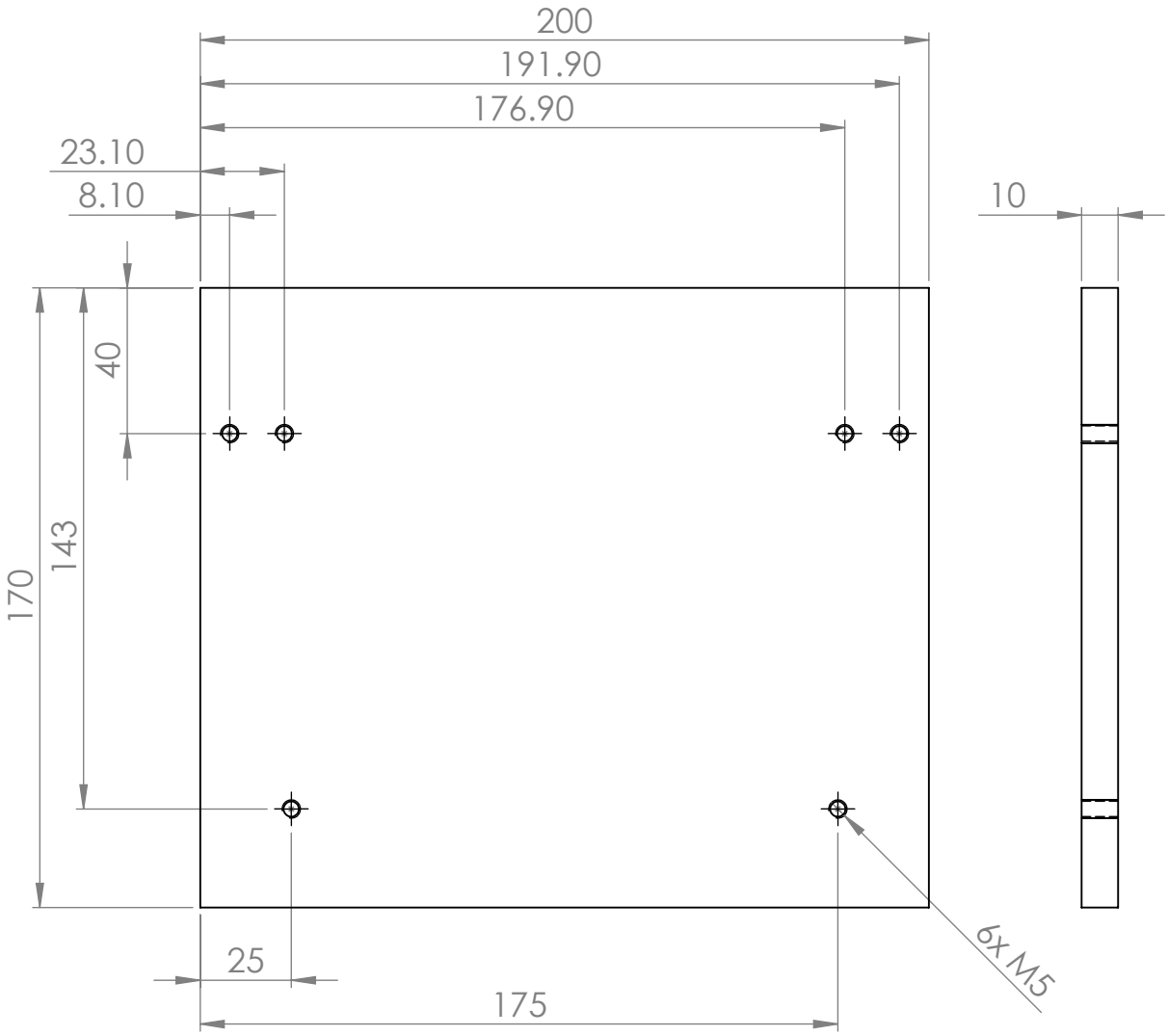
SOLIDWORKS Educational Product. For Instructional Use Only.



UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:		DEBURR AND BREAK SHARP EDGES		DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE				TITLE:
DRAWN							Chessboard connection to manipulator - part 2
CHK'D							
APPV'D							
MFG							
Q.A				MATERIAL:		DWG NO.	A4
				WEIGHT:		SCALE:1:1	SHEET 1 OF 1

SOLIDWORKS Educational Product. For Instructional Use Only.

RTU connection to tripod



UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:	DEBURR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING / REVISION

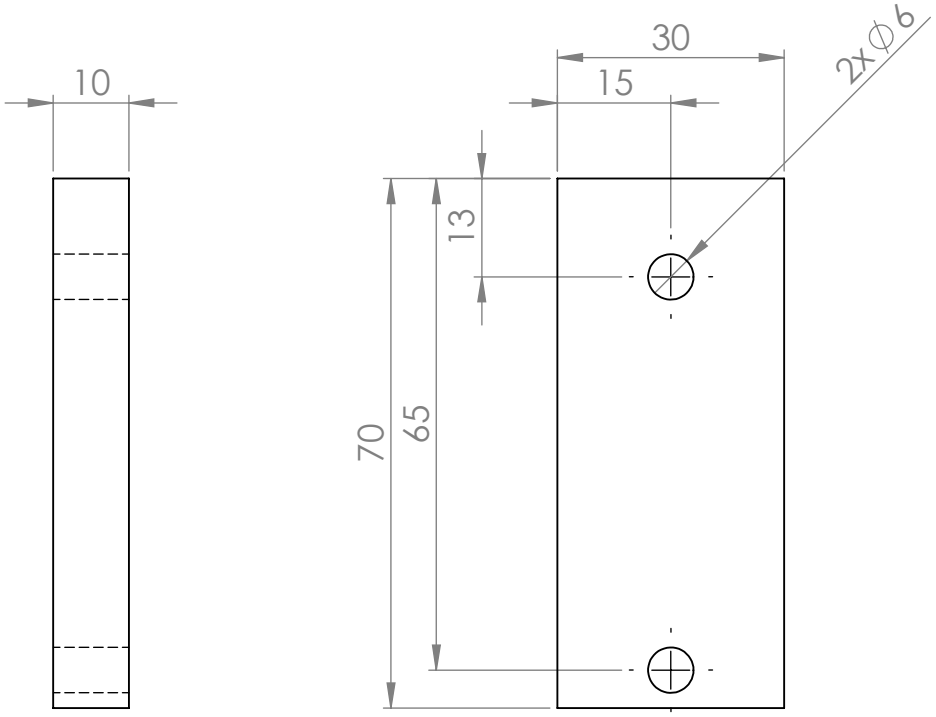
	NAME	SIGNATURE	DATE
DRAWN			
CHK'D			
APPV'D			
MFG			
Q.A			

TITLE:

RTU structure modifications

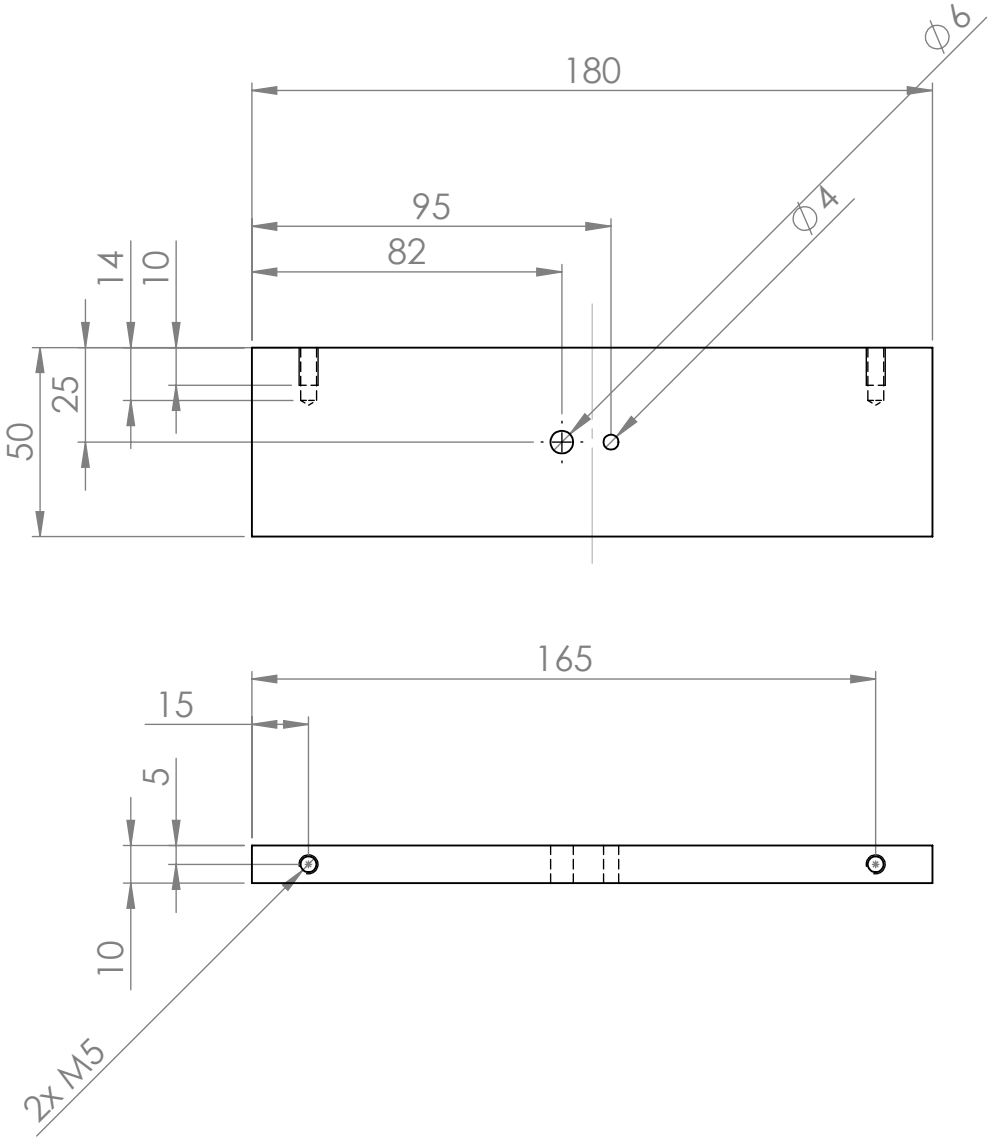
MATERIAL:	DWG NO.	A4
WEIGHT:	SCALE:1:2	SHEET 1 OF 1

SOLIDWORKS Educational Product. For Instructional Use Only.



UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBURR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION	
	NAME	SIGNATURE	DATE	TITLE: RTU connection to tripod - part 1		
DRAWN						
CHK'D						
APPV'D						
MFG						
Q.A				MATERIAL:	DWG NO.	A4
				WEIGHT:	SCALE:1:1	SHEET 1 OF 1

SOLIDWORKS Educational Product. For Instructional Use Only.



UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:	DEBURR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION		

	NAME	SIGNATURE	DATE			
DRAWN						
CHK'D						
APPV'D						
MFG						
Q.A						

TITLE:

RTU connection to tripod - part 2

MATERIAL:	DWG NO.	A4	
WEIGHT:	SCALE:1:2	SHEET 1 OF 1	

SOLIDWORKS Educational Product. For Instructional Use Only.