Visual Servoing with Collision Avoidance using Rapidly-exploring Random Trees

Miguel Gaspar Relvas do Nascimento

Thesis to obtain the Master of Science Degree in Engenharia Eletrotécnica e de Computadores

Supervisors: Prof. Alexandre José Malheiro Bernardino
Prof. José Alberto Rosado dos Santos-Vitor

Examination Committee

Chairperson: Prof. João Fernando Cardoso Silva Sequeira
Supervisor: Prof. Alexandre José Malheiro Bernardino
Member of the Committee: Prof. Rita Maria Mendes de Almeida Correia da Cunha

December 2019
Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.
Acknowledgments

I would like to thank Professor Alexandre Bernardino for the time and help given during the development of this thesis and for all the valuable feedback during the writing of this document.

I would also like to give a special thanks to Pedro Vicente for all the help, feedback and availability throughout this last months. His input was invaluable for the development of all the work that led to this report.

Lastly, but not least, I would like to thank everyone that was close to me during my time at university. Without you I couldn’t have done it. To each and every one of you – Thank you.
Abstract

Visual Servoing is a well known subject in robotics. However, there are still some challenges on the visual control of robots in robotic applications in human environments. In this thesis we propose a method for path planning and correction of kinematic errors using visual servoing. 3D information provided by external cameras will be used for reconstructing the environment and detecting the obstacles present. Rapidly-exploring Random Trees are then used to calculate a path through the obstacles to a given, previously calculated, end-effector goal pose. This allows for model-free path planning for cluttered environments by using a point cloud representation of the environment. The proposed path is then followed by the robot in open-loop. Error correction is performed near the goal pose by using real-time calculated image features as control points for an Image Based Visual Servoing controller that drives the end-effector towards the desired goal pose. With this method we intend to achieve the navigation of a robotic arm through a cluttered environment towards a goal pose with error correction performed at the end of the trajectory in order to mitigate both the weaknesses of Image Based Visual Servoing and of open-loop trajectory following. We made several experiments in order to validate our approach by evaluating each individual main component (environment reconstruction, trajectory calculation and error correction through visual servoing) of our solution.

Keywords

Resumo

Visual Servoing é um assunto bem conhecido em robótica. No entanto, ainda existem alguns desafios no controle visual de robôs em aplicações robóticas em ambientes humanos. Nesta tese, propomos um método para planeamento de caminhos e correção de erros cinemáticos usando servo visual. As informações 3D fornecidas por câmaras externas serão usadas para reconstruir o ambiente e detectar os obstáculos presentes. As Rapidly-exploring Random Trees são usadas para calcular um caminho através dos obstáculos a uma determinada pose de objectivo efectiva, previamente calculada. Isso permite o planeamento de caminhos sem modelo para ambientes com muitos objectos, usando uma representação em nuvem de pontos do ambiente. O caminho proposto é seguido pelo robô em malha aberta. A correção de erros é realizada perto da pose do objectivo, usando os recursos de imagem calculada em tempo real como pontos de controlo para um controlador de Visual Servoing baseado em imagem que direciona o end effector para a pose do objectivo desejada. Com este método, pretendemos alcançar a navegação de um braço robótico através de um ambiente desorganizado em direcção a uma pose de objectivo com correção de erro realizada no final da trajectória, a fim de mitigar os pontos fracos do Image Servo Visual Baseado em Imagem e da trajectória de circuito aberto a seguir.

Realizámos várias experiências para validar nossa abordagem, avaliando cada componente principal individualmente (reconstrução do ambiente, cálculo de trajectória e correção de erros através de Visual Servoing) da nossa solução.

Palavras Chave

# Contents

1 Introduction .................................................. 2  
  1.1 Motivation ............................................... 3  
  1.2 Objectives ................................................ 3  
  1.3 Outline ................................................... 4  

2 Background and State of the Art .......................... 5  
  2.1 Background ................................................. 6  
      2.1.1 Visual Servoing ...................................... 6  
          2.1.1.A Image Based Visual Servoing .................. 7  
          2.1.1.B Position Based Visual Servoing ............... 8  
          2.1.1.C Advanced Control Schemes .................... 9  
      2.1.2 Rapidly-exploring Random Trees (RRT) ............ 10  
  2.2 Related Work ............................................. 11  

3 Methodology .................................................. 13  
  3.1 Overview ................................................ 14  
  3.2 Obstacle Detection ....................................... 14  
      3.2.1 Filtering the environment ........................ 15  
          3.2.1.A Random Sample Consensus (RANSAC) .......... 15  
          3.2.1.B Removing the arms ............................ 16  
      3.2.2 Grouping the points into clusters ................. 17  
          3.2.2.A K-d trees and nearest neighbor search ....... 17  
  3.3 Collision-free trajectory ................................ 18  
  3.4 Visual Servoing .......................................... 19  
      3.4.1 Visual Features .................................... 19  
          3.4.1.A Scale-invariant feature transform (SIFT) .... 19  
          3.4.1.B Speeded Up Robust Features (SURF) .......... 20  
      3.4.2 Visual Servoing Controller ......................... 21  
      3.4.3 Controlling the robot .............................. 22
4 Implementation

4.1 Software framework

4.1.1 Robot Operating System (ROS)

4.1.2 Gazebo

4.1.3 MoveIt

4.1.4 ViSP

4.2 Software Architecture

4.2.1 Detection of obstacles

4.2.2 Collision-free trajectory

4.2.3 Visual Servoing

5 Experimental Results

5.1 Experimental Setup

5.2 Detection of obstacles

5.3 Collision-free trajectory

5.4 Visual Servoing

6 Conclusions

6.1 Conclusions

6.2 Future Work
Introduction

Contents

1.1 Motivation .................................................. 3
1.2 Objectives .................................................. 3
1.3 Outline ..................................................... 4
1.1 Motivation

Nowadays it is expected for robots to operate on human environments. However, these environments often have a high degree of clutter or other objects besides the ones we wish to manipulate or interact with. This makes it hard for a robot to execute a task in this environment since it doesn’t know how to avoid the obstacles while still reaching its goal.

Robot’s are extremely complex machines, with long kinematics chains which makes them hard to calibrate. Therefore, when executing a open-loop controller, the final pose of the end effector is often not the one desired by the user, due to internal model errors. Visual servoing has been employed as the method to solve the problem of open-loop controllers by closing the control loop with visual feedback.

However, visual servoing approaches don’t have a standard way to avoid obstacles present in the environment. Therefore, there is a need to calculate a trajectory through the environment while avoiding obstacles. Following trajectories with visual servoing in a cluttered environment can be very hard because of the high probability of occlusions. So how can we navigate an environment in open-loop but still have a high degree of confidence in the end effector’s pose at the end of the task?

1.2 Objectives

The objectives for this thesis can be divided in two parts: the calculation and execution of a collision-free trajectory through a cluttered work space and correction of end-effector final pose due to errors in the robot’s internal model using real-time calculated visual features.

Rapidly-exploring Random Trees [1] are a relatively well-known approach for path planning but its applications on robotics and cluttered environments are still limited due to the reliance on previously modeled work spaces. In order to build a more adaptable system we propose the run-time use of 3D information collected from a depth sensor overviewing the environment in which the robotic end effector will operate. Through this information the obstacles will be detected and sent to a path planning algorithm, along with a desired final pose, in order to calculate a collision-free trajectory. This will allow a greater flexibility in which this type of algorithm can be applied to, since it doesn’t require the offline acquisition of the environment 3D model. The only necessary equipment is an external depth sensor, allowing for a better adaptation to new world configurations. The calculated trajectory is then followed in open-loop in order to test its validity.

Visual Servoing is a common solution in robotics when it comes to closed-loop control. It is most often used as a way to correct the trajectory when there are external disturbances or when the robot model is inaccurate. In highly complex robots, these errors can make a given task impossible, especially on tasks that require precision, like grasping and manipulation of objects. Visual Servo control uses visual information as a proxy to the real state of the robot’s end effector and uses visual features from
these images to minimise an error function calculated through the current and desired values of these features. In this thesis to correct the end effector pose errors that come from the open-loop execution of the trajectory, we will use a visual servoing controller that uses real-time calculated visual features in order to take the robot’s end effector from its current pose to the desired one.

1.3 Outline

The organization of this document is as follows: in Chapter 2 a review of the theoretical background of visual servoing and RRT’S is given and the relevant related works are presented. In Chapter 3 an explanation of the methodology of this thesis and the theory behind it is given. In Chapter 4 the implementation details of the software produced in order to achieve the objectives proposed are explained, and the software frameworks used to achieve our goals are also briefly introduced. In Chapter 5 the experimental results for the solution proposed are presented. In Chapter 6 the conclusions we took from the developed work are given and the proposed future work to give continuation to this thesis is also proposed.
2 Background and State of the Art

Contents

2.1 Background ................................................................. 6
2.2 Related Work .............................................................. 11
In this chapter we are going to make a review of the theoretical background on visual servoing and Rapidly-exploring Random Trees. We are also going to go through some papers relevant to this thesis.

First, we provide a brief explanation of the basic approaches for visual servoing, Image Based and Position Based, as explained in [2]. Afterwards, we will present and discuss the current works available in the field of visual servoing. Finally, we will give a rundown of the main methods used for collision avoidance in visual servoing and their presence in the state of the art, and RRT.

### 2.1 Background

#### 2.1.1 Visual Servoing

The aim of visual-based control approaches is to minimize an error function $e(t)$,

$$e(t) = s(m(t), a) - s^*.\quad(2.1)$$

This is a general formulation that includes many different approaches. $m(t)$ is a vector of a set of image measurements (for example, coordinates for selected image feature points). These set of measurements is then used to construct the vector $s(m(t), a)$, where $a$ is additional information or parameters used to give further information on the selected measurements (like the camera’s intrinsic parameters). $s^*$ represents the desired goal values of the features.

Image Based and Position Based approaches select different vectors $s$. In Image Based the features are directly available on the camera image while in Position Based the features are a set of 3D points which need to be computed from image measurements.

Once $s$ is chosen a controller needs to be designed. The simplest controller is a velocity controller where the relation between the variation of $s$ and the camera velocity is given by

$$\dot{s} = Lsv_c,\quad(2.2)$$

where $L_s$ is the interaction matrix and $v_c = (v, \omega)$, $v$ being linear velocity and $\omega$ being angular velocity. Using 2.1 and 2.2, the final formulation of a controller is

$$v_c = -\lambda\hat{e}_ve,\quad(2.3)$$
where $\lambda$ is for exponential decrease of the error and $\hat{L}_e$ is the estimation of the pseudoinverse of $L_e = L_s$. In [2] a more detailed explanation is provided for the formulation of this controller.

### 2.1.1.1 Image Based Visual Servoing

Image Based Visual Servoing (IBVS) can be easily pictured as taking the initial position of a set of 2D image points and then "move" them towards a desired goal position on the 2D camera image.

![Figure 2.1](image.png)

*Figure 2.1: Example of a positioning task using IBVS: (a) desired camera position with respect to a target, (b) the initial camera pose, (c) the corresponding initial and desired image of the target [2].*

Traditional Image Based control schemes use the coordinates on the image plane to define a set of tracked points/features. We then need to relate the spatial velocity of the points in the view of the camera to their 3D velocity. This is done through an interaction matrix $L_x$. Below we can see the final form of this matrix and the relation between the 3D velocity of the points and its camera spatial velocity, where $p$ is a point in the camera's image defined by its $x$ and $y$ coordinates in the image plane. The full mathematical deduction of this matrix is presented in [2] (equations 6 through 11).

$$\dot{p} = L_x v_c, \quad p = [x \quad y] \quad (2.4)$$

$$L_x = \begin{bmatrix} -1/Z & 0 & x/Z & xy & -(1 + x^2) & y \\ 0 & -1/Z & y/Z & 1 + y^2 & -xy & -x \end{bmatrix} \quad (2.5)$$

The value $Z$ is the depth of the point relative to the camera frame and so any control schemes that use this interaction matrix need to make an estimation of this value. The camera intrinsic parameters are used to compute the 2-D coordinates of the points and so this interaction matrix cannot be used directly in 2.1 and needs to be approximated, since it is impossible to know perfectly in practice $L_x$. The more "classical" approximations for the interaction matrix $L_x$ can be seen in [2]. In this work we used the
approximation $\hat{L}_e^L = L_e^L$ where $L_e^L$ is the value of $L_e$ for the desired position $e = e^* = 0$. In this case the interaction matrix is constant since the value of the depth of the features is constant throughout the end effector’s motion. This is because only the desired depth of the features needs to be set.

When the target configuration is too far from the current configuration, a pure IBVS approach may be attracted to local minima or cross a singularity of the interaction matrix [3] and the camera motion may have unplanned or undesired movement, since this method only minimizes 2D image metrics. IBVS also needs a robust estimation of depth, since poor estimation may lead to instability or induce errors in the trajectory. Another problem is that if the visual features are occluded or the image features leave the camera’s view, the visual controller stops working. This controller does not control the various parts of the robot, which may lead to collisions with elements of the environment.

IBVS is more popular in Eye-in-Hand (camera attached to the end effector) configurations because the goal of this method is to take some features from an initial position in the camera image (and their estimated depth) to a goal position. If the camera is attached to the end effector, the camera will have the same movement as the end effector, making the servoing task much easier since the error between the initial and final position of the features in the camera image is directly correlated to the motion of the servoing.

### 2.1.1.B Position Based Visual Servoing

While IBVS is a purely 2D approach, Position Based Visual Servoing (PBVS) is a purely 3D approach. It uses the relative camera pose with respect to some reference coordinate frame. To get this pose from some image measurements the camera intrinsic parameters and the 3D model of the observed object need to be known. From the camera pose the image features $s$ used in the control of the servoing are selected. These image features $s$ can be defined in different ways, leading to different PBVS schemes.

In [2] we can see two different ways to define these features, leading to two different controllers which will lead to different camera motion and image trajectories. One possible way to define these pose features is to define them relative to the object frame which leads to $s = (^c t_o, \theta_u)$, $s^* = (^c t_o, 0)$ and $e = (^c t_o - ^c t_o, \theta_u)$, where $^c t_o$ and $^c t_o$ give the coordinates of the object frame in relation to the current camera frame and desired camera frame, respectively. Moreover $\theta_u$ is the angle/axis parameterization for the rotation. The interaction matrix related to $e$ is given by

$$ L_e = \begin{bmatrix} -I_3 & [^c t_o]_x \\ 0 & L_{\theta_u} \end{bmatrix}, $$

in which $I_3$ is the $3 \times 3$ identity matrix and $[^c t_o]_x$ represents the skew-symmetric matrix associated to the translation vector $^c t_o$ and
\[
L_{\theta u} = I_3 - \frac{\theta}{2}[u]_\times + \left(1 - \frac{sinc\theta}{sinc^2\frac{\theta}{2}}\right)[u]^2._\times.
\]

By using (2.3) we get the control scheme \( v_c = -\lambda \tilde{L}_{e}^{-1}e \), where

\[
\tilde{L}_{e}^{-1} = \begin{bmatrix} -I_3 & [t_o]_{\times}L_{\theta u}^{-1} \\ 0 & L_{\theta u}^{-1} \end{bmatrix}.
\]

We get after simple developments

\[
\begin{cases}
\dot{v}_c = -\lambda(\dot{c} \times t_o - \dot{c} \times t_o) + [\dot{t}_o]_{\times} \theta u \\
\dot{\omega}_c = -\lambda \theta u
\end{cases}
\]

Another possible way to define \( s \) is \( s = (c^* t_o, \theta u) \), a definition in which the features are defined relative to the desired camera frame instead of the current camera frame as was done in the previous example, resulting in \( s^* = 0 \) and \( e = s \). Therefore \( L_e \) is given by

\[
L_e = \begin{bmatrix} R & 0 \\ 0 & L_{\theta u} \end{bmatrix}.
\]

The decoupled translational and rotational motions allow the simple control scheme of

\[
\begin{cases}
\dot{v}_c = -\lambda R \dot{c}^* t_o \\
\dot{\omega}_c = -\lambda \theta u
\end{cases}
\]

Since the features in PBVS are represented in 3D, a theoretical optimal trajectory can be achieved but the slightest error in image measurements can lead to an error in pose estimation and since the control is done through pose computation and not through the camera image, there is no way to ensure that the object will stay in the camera’s view [3]. Furthermore, if the 3D models have big errors it will also lead to pose estimation errors, affecting the servo trajectory. This type of visual servoing also has some of the problems of IBVS regarding the visibility of features and lack of collision avoidance.

While IBVS is more suited for Eye-in-Hand configurations, PBVS is most common in Eye-to-Hand approaches (camera looking to the end effector, like in humanoid robots). This is because PBVS’s goal is to get the end effector to a desired final pose by getting constant feedback on the 3D pose of the end-effector, therefore it’s only natural that a 3D pose estimation reliant method like PBVS is most popular in a camera configuration that allows the computation of end-effector and object pose.

### 2.1.1.C Advanced Control Schemes

We will now briefly describe some more advanced visual servoing control schemes that try to mitigate the drawbacks of image and position based, while taking still maintaining some of their advantages.
Hybrid Visual Servoing [4] or 2-1/2 D Visual Servoing is a method that combines IBVS and PBVS into a single control law. This approach uses mixed 2D and 3D features (this why it is called a 2-1/2 D approach) in order to decouple the rotational and translational parts of the end effector’s motion using, for example, the angular velocity control from PBVS and doing cartesian velocity control with IBVS. This type of approach allows for a great degree of freedom on the type of features that can be chosen.

This hybrid approach inspired some researchers to develop a controller that also decouples rotational motion from translational motion but it takes its features only from image information, trying to find features with the desired decoupling properties [5]. From [5] stemmed switching schemes, which are controllers that choose when to use an IBVS or PBVS controller, according to some switching criteria.

### 2.1.2 Rapidly-exploring Random Trees (RRT)

Rapidly-exploring Random Trees (RRT) is a rapid search algorithm first described in [1]. It is based on random sampling, storing the collected random samples in a tree structure and continues its search until it reaches the target. Both the target and the obstacles need to be known a priori.

![RRT algorithm path extension process.](image)

In Figure 2.2 it is presented the extension process for this path planning algorithm. The root node of the tree is the initial point for the path, \( x_{\text{start}} \). To extend the tree a random state vector \( x_{\text{rand}} \) is calculated and the nearest state vector \( x_{\text{near}} \), in the state space of the considered environment, to \( x_{\text{rand}} \) is found. \( x_{\text{near}} \) will then be the endpoint for this extension of the tree. If a collision was detected in the extension process, this extension will stop and a new \( x_{\text{rand}} \) will be calculated. This process is repeated until \( x_{\text{goal}} \) is reached.

This describes the base formulation of this algorithm but there are different implementations. One of those is RRT* [6]. While base RRT finds a trajectory, RRT* optimizes the distance traveled on this trajectory. While it extends the tree it checks the cost (distance) of each neighboring node, choosing the one with less cost. This results in a path that travels less distance but requires more computation.

Some algorithms tried to improve on RRT*. One of these is Informed RRT* [7] which tries to limit the search space used to grow the search tree, avoiding unnecessary expansions of the tree. When an initial solution is found the search space is limited to an ellipse that contains all possible improvements...
on the current best solution. When a better solution is found a new ellipse is also calculated. In Figure 2.3 we can see a comparison between searched space states when using RRT* and Informed RRT*.

![Figure 2.3: Search space comparison on similar cost solutions using RRT* and Informed RRT* [7].](image)

### 2.2 Related Work

Now that we gave a general outline of the basic methods used in visual servoing and an explanation of the RRT path planning algorithm and its variations, we will go into the current works in the field of visual servoing and the application of RRTs in this subject.

In [8] the authors define three kinds of approaches for motion planning in robotic grasping and manipulation. The first of these approaches is **sense-plan-act**. This kind of systems are built through strong modularization where perception feeds a model of the environment to a motion planner which finds a collision-free path and is then tracked by a stiff and accurate controller. This system has the advantage of dividing the problem into easier subproblems and its more suited to well-structured environments but copes less well with uncertainty and changes in the environment. To help solve this exists **sequential sense-plan-act** where the robot does not only request for feedback at the beginning of the task, but also at pre-defined moments (by the application’s developer or user) during execution. The second of these approaches is **Locally Reactive Control** where only the local geometry near the current end effector pose is taken into account. Visual Servoing is one type of locally reactive controllers closing the control loop around visual information but that has limitations in more complex tasks that require planning. Finally, the last category is **reactive planning** which is a combination of the two approaches explained above, allowing for local control and motion planning but there are few solutions applied to manipulators with high number of degrees of freedom. Our work inserts itself in between **sense-plan-act** and **Locally Reactive Control** since it uses components from both elements, namely by first doing a overall representation
of the environment and computing a collision-free path and then using a locally reactive controller to
 correct end effector pose errors.

Current applications of the RRT algorithm to the field of Visual Servoing are quite limited. In [9] a
 rapidly-exploring random tree is grown in the camera state space while being simultaneously tracked
 in the robot joint space. This work uses an IBVS controller in a Eye-in-Hand robot configuration and
 requires a previously 3D model of the operating environment in order to calculate the collision free
 trajectory. The image feature positions in the camera image are then tracked through the trajectory,
 effectively "drawing" their trajectories in the camera’s image. This work has some drawbacks since it
can not be adapted to new environments without first modeling said environments and the image visual
features can not suffer occlusions from objects in the environment.

Visual servoing is a powerful tool since it allows for the correction of the propagated errors in the
robot’s internal model, closing the control loop. Most works that use an IBVS approach tend to use
markers in order to quickly calculate the image features to be tracked in the control loop. Although this
presents good results, these markers require extra work and reduce the adaptability of the systems
that use them, especially when it comes to grasping daily objects. In [10] SURF features are used with
an IBVS controller. Their approach uses these features to first compare a query image of a certain
object and the current image on the camera. By doing feature matching on both these images, the
algorithm selects a Region of Interest which will be tracked through the camera’s motion. The object will
be in this region and this is done to reduce the time spent on calculating and comparing features. After
this process they calculate the geometrical features, like area of the region of interest and its center,
that have a correlation with change on each of the six degrees of freedom. By doing this, each of the
features is decoupled from the rest.

In [11] we can see a Eye-to-Hand PBVS controller where the visual feedback obtained from the
cameras in the robot’s eyes is used to calibrate the robot’s internal model [12] [13] in order to mitigate
the errors in the inverse kinematics of the robot. This is done by measuring the offset $\beta$ between the
robot’s real joint angles and the measured robot’s joint angles (as perceived by the internal model). This
is represented by $q^r = q + \beta$ where $q^r$ are the real angle values and $q$ are the measured angle values.
This allows for a better end-effector pose estimation. The values for $\beta$ are estimated by comparing the
images captured by the cameras with images generated by a game engine and the CAD model of the
robot. The PBVS controller used is similar to the one described in [2].
3

Methodology

Contents

3.1 Overview ................................................................. 14
3.2 Obstacle Detection .................................................... 14
3.3 Collision-free trajectory ............................................. 18
3.4 Visual Servoing ......................................................... 19
In this chapter we will explain the methodology used to develop the proposed work. The theoretic concepts of the tools used to do so are going to be explained and how they are integrated into our approach.

3.1 Overview

In this chapter the method to achieve what was proposed in this thesis will be explained. On Figure 3.1 is shown the approach pipeline for the proposed solution. Through 3D information we get an approximate representation of the environment that is updated when relevant changes occur. Afterwards, obstacles are detected in this representation. Then a collision-free trajectory is calculated using the Rapidly-exploring Random Trees algorithm with the given initial and final poses for the end-effector. The trajectory is then followed in open-loop to the vicinity of the object. The errors present in the robot model are propagated throughout the robot arm's motion and the actual pose of the end-effector will not be the desired one. These errors are then corrected through Visual Servoing using a reference image previously taken. By doing visual feature matching between this reference image and the current image on the end effector's camera we get the feature points to be used in the visual servoing controller.

![Figure 3.1: Schematic of the planned approach.](image)

3.2 Obstacle Detection

To achieve the proposed task it is first needed a representation of the environment in which we desire to operate. This representation is obtained through a series of operations on a captured point cloud from an external RGB-D camera, which are cameras that enable the perception of depth in images. In this
an explanation of these operations will be given and how they contribute to detect the obstacles present in our work space.

3.2.1 Filtering the environment

The first task to be performed is filtering out the parts of the environment that are not to be considered as obstacles to our task.

We take as an assumption that our work space is a table with objects on it, one of which is the one to grasp, which we are given the desired pose in order to grasp it. Therefore, in order to identify the objects only we need to filter out the table and the arms of the robot.

The removal of the robot’s arms may not seem an obvious necessity for the success of our task but it is. Since they are the only moving parts in our scene and the ones we want to avoid collisions between them and the environment. If the arms are not remove/filter out from the point cloud, the algorithm will represent the arms as an obstacle. Thus, the robot will not be able to move since the arms will be (supposedly) colliding with some environmental obstacles (i.e., themselves).

Below we explain the main method to filter out the table in order to only have point cloud points that represent either objects or visible parts of the robot. An explanation on the method to remove the points the represent part of the robot’s arms is also given.

3.2.1.A Random Sample Consensus (RANSAC)

To remove the points that represent the table the algorithm used was Random Sample Consensus (RANSAC) [14]. In a dataset where there are inliers (data that fits a given model) and outliers (data that does not fit said model) RANSAC uses a voting scheme where each element of data votes on multiple models. This algorithm takes as an assumption there are fewer outliers then inliers and so the outlying data will no consistently vote for any single model and that enough features present in the inliers in order for them to “agree” (through voting) on a model.

The RANSAC algorithm functions by iteratively repeating a two-step process. The first step is to compute a fitting model and its parameters using a minimal sample from the dataset (hypothetical inliers). The number of data elements is only large enough to estimate those parameters and the elements are chosen randomly from the complete dataset. On the second step the algorithm checks which elements of the complete dataset fit the model previously computed on the first step. These elements are classified as outliers or inliers to this estimated model if they fit or not the model being evaluated in that iteration of the algorithm with a certain margin of error, given a certain loss function according to a specific model.

The set of inliers of this proposed model is called the consensus set. The process explained above is repeated until this set of inliers has a certain size. After this the model may be improved by being
recalculating it using the members of the consensus set.

On Figure 3.2 we can see a quick example of the effect that this algorithm has on a simple 2D dataset where the objective is to find the line that fits the most number of points. If a method that did not classify outliers and inliers was used, like the least squares method, a line would be calculated but it would be a bad fit since it would take into account the outliers in addition to the inliers.

![Figure 3.2: Inliers and outliers on a linear model estimated with RANSAC and with linear regression.](image)

We use the RANSAC algorithm to identify a planar model. This means that the algorithm will try to find the inliers that fit a certain proposed plane equation. These results in the identification of the table top part which is the predominant plane in our environment. By knowing the points that belong to this plane (the inliers that RANSAC finds) we can then filter them out from our point cloud of the environment. We then have some scattered groups of points that represent objects or parts of the robot.

### 3.2.1.B Removing the arms

Now that the table has been filtered out, the only parts that still remain in the point cloud that we do not want to be perceived as objects is the arms of the robot, as been explained in the beginning of this section. The best way we have to identify what parts of the point cloud belong to the robot's arms is through its own perception of its body, the robot's internal model.

Through the robot's internal model we can know the location of each of its arm joints. A line is then "drawn" between each consecutive joint. If a point is inside a certain radius from this line, meaning if it is inside a cylinder with a certain radius where the axis of rotation is the line between two consecutive joints, the cluster to which that point belongs is seen by our system as representing part of the arm in the environment point cloud (the process of grouping up points into clusters is explained further into this chapter).
On Figure 3.3 we can see the results of the complete environment filtering method with the joint to joint lines drawn. In the filtered point cloud there are still parts of the environment that are not objects (the front strip of the table and a piece of the robot) but these elements do not intervene in the identification of the objects since they are clearly separated.

![Figure 3.3: (a)- Original point cloud of the environment. (b)- Point cloud of the environment after table and robot’s arm filtering. The blue lines represent the connecting lines between each consecutive arm joint.](image)

3.2.2 Grouping the points into clusters

After getting a filtered point cloud from the method explained above we need to group up the points into object clusters so that our system can clearly separate and identify each one of them to pass them as obstacles to the next step of our pipeline. We group up the points into clusters based on their proximity to one another. After assigning all the points into clusters we have our representation of the obstacles and so we can plan a trajectory with the methods already approached. In this section we will explain the method used to do this point clustering.

3.2.2.A K-d trees and nearest neighbor search

In order to make clusters out of the random points in our point cloud first it was necessary to represent the data in a way that would make it easier to make a search. The representation used in this approach was the construction of a k-d tree.

K-d trees are a type of binary trees in which each node represents data with $k$ dimensions. In this case we have data with 3 dimensions representing the $x$, $y$ and $z$ coordinates of the points. To explain the organization of these type of binary trees we will use an example. On Figure 3.4 we can see an already constructed k-d tree with some nodes in it. The nodes in this tree are two-dimensional in order to be easier to understand and explain the method used to construct them.
When putting a node in the tree, a splitting axis is chosen to be associated to that node. This axis will then create two hyperplanes that split the space in two. To the left of the node being inserted will be the nodes where the value in the chosen axis is lesser than the parent node and to the right are the values greater than the parent node. In the example given we can clearly see this. In the root node the splitting axis chosen as $X$ and so to the left of the root are only nodes with a $x$ coordinate value lesser than 7 and to the right are the nodes with this value greater than 7. We can see this process repeat itself on the second level of the tree but this time with the $Y$ axis, as in the construction of the tree is necessary to cycle through the splitting axis. If we were to insert a node with coordinates (10,7) it would be a right-side child node of the node (9,6) ($x$ greater than 7 and $y$ greater than 6).

After having this representation of our point cloud we can then begin a nearest neighbour search in order to group up the points into clusters. Having the data organised as a k-d tree greatly improves the performance of the search algorithm since it eliminates big portions of the dataset during the search. This search begins by setting up an empty list of clusters and a queue of points to be checked. Then for a given test point in the queue the algorithm searches for every other point inside a sphere with a given radius of said point. When a neighbor is found (a point inside the sphere) it is added to the queue of identified neighbors if it is not already in there, meaning that it will be checked for neighbors in a later iteration. After all points in the queue have been checked they are pushed into the list of identified clusters. The algorithm ends when all points of the tree have been assigned into a cluster.

**3.3 Collision-free trajectory**

To calculate the trajectory through the detected obstacles we use the path planning algorithm’s implementations that are available in the Open Motion Planning Library through the MoveIt planning framework. Both of these tools are explained further in Chapter 4.
The collision-free trajectories are planned in 3D space. What this means is that the planners build the trajectory through a series of pose waypoints and then, through inverse kinematics, an arm configuration that doesn’t produce collision with the rest of the environment is found. The pose waypoints are calculated through sampling of the 3D workspace.

In the context of RRT’s this means that the state space in which the tree is grown is the end effector pose state space. In each iteration of the tree growing process, the nearest pose state to the current one is found. Then the path necessary to reach them is tested for environment collisions and joint limits. If the path passes these tests, it is added to the tree.

3.4 Visual Servoing

Now that we have a calculated trajectory and we have executed it with an open-loop controller, namely by using the robot’s self perception through its internal model, we need to correct the errors propagated throughout the motion of the robot’s end effector because the robot’s kinematics are extremely complex and therefore have some degree of errors or miscalculations in it. To correct these errors we use an image based visual servoing controller. With a reference image of the environment taken with the target end effector pose and current images from the camera attached to the robotic manipulator, drives the end effector from its current pose to a desired one. However, to execute this visual servoing task we first need to define the image features used to calculate the error between current and desired camera pose. In order to not use artificial markers that would not be present in a day to day scene we use the feature detecting and matching algorithm SURF [15] to calculate the common image features and match them between the reference and current camera images.

Below a description of the SIFT [16] [17] algorithm, the method in which SURF was inspired, and of SURF is given. A description of the visual servoing controller used is also presented, as well as an explanation on how the camera velocities that the controller calculates were translated into end effector motion.

3.4.1 Visual Features

3.4.1.A Scale-invariant feature transform (SIFT)

Scale-invariant feature transform (SIFT) [16] [17] is a feature detection algorithm that finds and describes local features in images. These features are mostly used to find a specific object in an image with clutter and where the environment can be subject to changes like illumination, contrast or scale. This done by matching the features found in a training image and the features present in some test images.

The SIFT algorithm works by first detecting keypoints, which are points of interest in the image.
Keypoints are the maxima and minima of the Difference of Gaussians (DoG) taken when a number of Gaussian filters with different scales is passed on the image. A DoG image is given by

$$D(x, y, \theta) = L(x, y, k_i\theta) - L(x, y, k_j\theta)$$  \hspace{1cm} (3.1)

where \(L(x, y, k\theta)\) is the convolution of the original image with a variable-scale Gaussian \(G(x, y, k\theta)\), where \(x\) and \(y\) are the image coordinates. Therefore, the operation described in 3.1 is the difference of two images with Gaussian convolutions at different scales \((k_i\theta\) and \(k_j\theta)\). The Gaussian images are grouped up according to their scale and then DoG is performed on images with similar scales.

To find the local maxima and minima of the DoG images, every pixel is compared to its eight numbers on the same scale image and to its nine neighbors on the adjacent scales. If the pixel's value is the maximum or minimum among those compared to it, it is then selected as a possible keypoint.

Many keypoint candidates are found through this method but not all of them will provide a valid keypoint. In order to find the best keypoints a detailed fit to nearby data is done, thus eliminating points with low contrast or that poorly localized along an edge. The accurate position is found by using the quadratic Taylor expansion of the DoG function using the candidate keypoint as the origin. The location of the extremum is calculated by taking the derivative of the Taylor expansion and setting it to zero. If the offset between the candidate point and the location of the extremum is greater than 0.5 the extremum is closer to another candidate keypoint. The process is then repeated for this new candidate. Otherwise, the offset is added to the keypoint in order to get its interpolated estimated position.

The Difference of Gaussians function has a high response along the edges, even if the keypoint does not provide flexibility when noise is present and this is necessary to remove this high responses in order to improve stability.

After finding the location of the keypoint and filtering out the unstable candidates, an orientation descriptor is assigned to them. The orientation is calculated by the local image gradient directions. This allows for invariance to rotation since the keypoint is described by local changes in its vicinity.

The previous operations gave the keypoint invariance to image location, scale and rotation. The final step of the algorithm is to assign a descriptor that provides invariance to other variable conditions like lighting and 3D viewpoint. A descriptor is created by computing the gradient magnitude and orientation at sample image points around the keypoint. These samples are then condensed in a orientation histogram that summarize off the subregions around the point.

### 3.4.1.B Speeded Up Robust Features (SURF)

Speeded Up Robust Features (SURF) \cite{15} is a local feature detector inspired by the SIFT algorithm described above but with an improved computational performance. However, while SIFT calculates that Difference of Gaussians in rescaled images, SURF uses square-shaped filters as an approximation
to Gaussian smoothing because using square filters is much faster when using a integral image (a representation of the sum of values in a rectangular area of a grid). SURF uses a blob detector based on a Hessian matrix to find keypoints. Keypoints are found where the determinant of this matrix is maximum.

To locate the points of interest, the SURF algorithm uses Gaussian filters of increasing size and then performs non-maximum suppression in a neighborhood of the keypoint candidate over scales in order to locate it.

In order to assign an orientation feature to its descriptor, the Haar wavelet response is calculated in the $x$ and $y$ directions, on the image's referential, in a radius proportional to the scale in which the point of interest was detected. The results are then plotted and the dominant orientation is calculated by summing the responses within a defined sliding window. As for the descriptor a similar process to the the SIFT descriptor is done but instead of using gradient magnitude and orientation, SURF uses the response to Haar wavelet.

### 3.4.2 Visual Servoing Controller

Now that we have described the method with which we calculate the features used in the servoing we can talk about the visual servoing controller and how it uses said features.

Firstly, we take the reference image and the current camera image and perform feature detection on each one of them. We then compare the descriptors from each of the images using a k-nearest neighbors algorithm in order to find the matches between features in the two images.

Although this method provides matching between the descriptors of the keypoints in the two images, some of these matches are in fact "false" matches, meaning they are matches that do not represent the same area of the environment in the camera image. To filter these bad matches we use Lowe's Ratio Test [17]. This method takes a certain keypoint from the reference image and its two best matches from the current image (since the nearest neighbors method we used as the $k$ parameter set to 2) and looks at the ration of the distance metric of these two matches (the distance metric can be seen as a scoring mechanism that indicates how close a keypoint is from another). If this ratio is above a defined threshold, this keypoint match is discarded. It is discarded because failing this test means there was another keypoint on the current image which was close to being matched to the keypoint on the reference image and therefore the pair of matched keypoints does not provide a reliable connection between the two images in that area.

After filtering out the unreliable matches we get a set of dependable keypoint parings. This set of features are then used by our controller as the current and desired features, depending if they belong to the reference or current camera images. The controller we used is a velocity controller as the one shown in 2.3. As for the construction of the interaction matrix of image based visual servoing, shown in Eq.
2.5 We chose to define it in relation to the desired image visual features. By doing this, we don’t have to continuously estimate the depth parameter of the features. This is because the interaction matrix is constructed in relation to the desired features which, obviously, don’t change during the servoing task. As for the $x$ and $y$ values in the matrix, which are the image plane coordinates of the point, they are calculated using the camera’s intrinsic parameters and the known pixel coordinates of the SURF features by using the equation in 3.2 where $(u,v)$ are the pixel coordinates of the feature point, $c_u$ and $c_v$ are the image principal point pixel coordinates and $f$ is the focal length.

$$\begin{align*}
x &= (u - c_u)/f \\
y &= (v - c_v)/f 
\end{align*} \quad (3.2)$$

By doing the process described above we can take SURF matched keypoints from the current and reference image and pass them as the desired and current features to our IBVS controller. The controller then calculates the camera velocity that will minimize the error between said features.

### 3.4.3 Controlling the robot

Now that the velocity to minimize the error between features of the visual servoing controller has been calculated we just only need to make the robot move the end effector with said velocity. However, robots can not operate directly through a Cartesian velocity to be applied to the end effector, this velocity needs to be translated into a set of joint velocities that produce that desired change in pose of the end effector.

In a six Degrees of Freedom (DoF) robot this process would be relatively simple. A Jacobian matrix is a matrix of the partial derivatives of a certain vector function. When it comes to robots, this matrix translates the velocity of a given joint into a certain Cartesian space change through 3.3, where $v$ represents 6 DoF velocity, $J$ is the robot’s Jacobian matrix and $\dot{q}$ is the vector of joint velocities.

$$v = J\dot{q} \quad (3.3)$$

With a non-redundant manipulator, in order to know the joint velocities that correspond to a 3D space velocity we would only need to invert the Jacobian matrix and multiply it by the velocity vector in column matrix form. But with a manipulator with seven or more degrees of freedom (a redundant manipulator) the Jacobian matrix is a $6 \times 7$ matrix which is obviously non-invertible (in our case since our robot has 7 DoF). To solve this we take the approach presented in [18], where the Moore-Penrose pseudo-inverse of the robot Jacobian is used (represented by $J^+$). To calculate the pseudo-inverse the Singular Value Decomposition (SVD) was used 3.4.

$$A = U\Sigma V^*, A^+ = V\Sigma^+ U^* \quad (3.4)$$
Now that we have a generalization of the inverse of a non square matrix we can calculate the joint velocities that translate into the desired velocity by applying 3.5.

\[ \dot{q} = J^+ v \]  

(3.5)

Having the desired joint velocities we can finally execute the velocity command given to us by the visual servoing that will reduce the error between the calculated SURF features. This process will eventually lead to having the desired pose of the end effector, thus correcting any error there was in its final pose after the execution of the collision-free trajectory.
## Implementation

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Software framework</td>
<td>25</td>
</tr>
<tr>
<td>4.2 Software Architecture</td>
<td>26</td>
</tr>
</tbody>
</table>
In this chapter we will go into detail about some details of the implementation of the proposed approach. First, a rundown of the used software and platforms is given in order to better some of the choices done during the implementation. After this introduction to the software platforms used we will go through each main component and explain the important details and code structure of the implementation.

4.1 Software framework

4.1.1 Robot Operating System (ROS)

ROS [19] is an open-source robotics middleware that collects several software frameworks in order to allow software development for robots. It implements various tools and libraries that facilitate creating robotic applications due to its abstraction of hardware and device drivers. It also gives a great degree of flexibility and robustness since its communication structure allows for cross-language operation and an application developed for one robot can be easily adapted for a different robot.

The two most important concepts in ROS are packages and nodes. Packages are the file system structure in which ROS applications are divided. They can contain messages, services or libraries and ROS packages can use something implemented by another package. One way for packages to communicate with each other is through nodes. ROS nodes are the software modules of any given ROS application. Packages can have a multitude of nodes within them and they communicate with each other sending and receiving messages sent through ROS topics. Nodes can subscribe or publish to topics, receiving and sending messages through them.

One of the most useful services implemented in ROS, and one that we used extensively in this thesis, is the TF tree. This service basically publishes the transforms of every coordinate frame present in your system in a tree structure. For example, if we want the coordinate transform from one of the robot's arm joints to the robot's base joint, this package would go from the robot's arm joint, to the robot's shoulder frame and then to the robot's base frame, backtracking its way from one frame to another since each consecutive frame is connected in the tree with the transform to each of its downstream "child" frames.

In our work we used the ROS Kinetic Kame distribution.

4.1.2 Gazebo

To simulate our environment and the robot's behaviour to our solution we used the Gazebo simulator [20] since the robot used had a Gazebo model and ROS already has a package that allows for direct interaction between it and the simulator through ROS topics, allowing for control of the robot's model. This enables testing our application inside a controlled simulated environment.
To use Gazebo a set of URDF (Unified Robot Description Format) and SRDF (Semantic Robot Description Format) files that describe the simulated world and the robot need to be given to the simulator. The robot is defined by a set of rigid bodies and the joints that connect them. In conjunction with a physics engine, we can give velocities to the robot’s components and visualize its interaction with the simulated world. Gazebo also simulates cameras and other sensors commonly used in robotic applications, whether they are part of the robot or not.

4.1.3 MoveIt

In order to perform collision-free path planning we use the MoveIt motion planning framework [21]. MoveIt runs on top of ROS and its available through a ROS package and incorporates many internal and external packages.

Through the planning scene, which represents the areas occupied and unoccupied by obstacles, and the move group, with the robot’s moving parts being controlled and the user’s planning requests, the framework calls its planning algorithms. Moveit performs planning through an external library called Open Motion Planning Library (OMPL) [22], which is a motion planning library with several planners implementations but without the notion of a robot and so MoveIt provides the back-end computation necessary for problems in robotics. The RRT implementation used in this work comes from this library. After calculating the path MoveIt sends the necessary commands to the robot’s controllers and executes the desired task. This works also with the simulated controllers available though the Gazebo simulator.

4.1.4 ViSP

ViSP [23] stands for Visual Servoing Platform and it’s a library that allows the development of applications with visual tracking and visual servoing by computing control laws that can be applied to robotic systems.

The visual servoing control law used in this work that calculates the end effector velocity that minimizes the error between present and desired visual features is implemented through the functionality of this library.

4.2 Software Architecture

4.2.1 Detection of obstacles

We will begin by explaining the code implementation of the process described in the previous chapter that allows for obstacle detection.

We start by launching our gazebo world with our robot, a table with a number of objects on top of it and a depth sensor pointing at the general direction of the table in order to capture its entirety (or at least
the part in which we wish to operate). We take the information that the sensor reads from the ROS topic /camera/depth_registered/points by subscribing it in our cloud_transformer ROS node. What this node does is taking that information, that comes in the depth sensor frame and transform it to the world, which is going to be our reference frame throughout. This transform is grabbed from the TF tree. By doing this we now have the depth sensor information represented in our common frame, which facilitates its use later its visualization through Rviz. We then send the transformed point cloud to a ROS topic we create called /obj_recognition/point_cloud.

Then our obj_recognition_segmentation ROS node takes this point cloud and a series of operations will be done through the Point Cloud Library (PCL) [24]. This library will be our main way to operate in point clouds. The first operation we perform is applying a pass-through filter to remove the table's legs. This is done to simplify the visualization of the results, since having the table legs in the point cloud would not affect the functioning of our program.

After filtering out the legs we apply RANSAC in the resulting point cloud with a planar model and an adjustable distance threshold (to classify model inliers). This removes the table's top from the point cloud. Instead of discarding the filtered points we save them in their own point cloud since we want the top part of the table to be perceived as an obstacle by our application but we still need to remove it to identify the obstacles.

Now we have a partial point cloud of the original scene. This point cloud has points belonging to the environment parts that we didn't filter, this is, the obstacles and the robot. Clustering through kd-trees is done in order to group up these points into clusters. The criteria to define the cluster are cluster minimum and maximum size and the distance tolerance (the radius of the sphere used to group up points). This values can be adjusted according to the specific needs of a dataset. After clustering the data, we filter out the last part of the point cloud that is not an object and that will cause problems to our application if is perceived as such: the robot's arms. As explained before, to do this we draw lines between each consecutive joint frame on a arm of the robot. This is done with the aid of the TF trees, which we use to transform the origin of each of those frames into our world frame. Doing this transformation allows for comparison between these lines and the point cloud. For each point in a given cluster we call the function checkCylinder.

This function takes the origin of two consecutive frames, one test cluster point, the distance between them and the radius around the line formed by these two frames. This radius represents what we perceive to be the area around each line that belongs to a part of the robot's arm. With this information the function creates two vectors, one that goes from one frame to the other and one that goes from the first given frame to the test point. The dot product of these two vectors is done and compared. If it's smaller than zero or greater than the squared length between each frame then it means it's either behind the "base" cylinder cap or outside the cap of the "top" cylinder cap (this cylinder we talk about is

27
the volume formed around the line between the two frames and the radius around it). If the test point is between the two caps, it is then calculated the distance between it and the cylinder axis line. If it is greater than the given radius it means it doesn’t belong to the arm, if it is smaller then it is part of the robot’s arm and the cluster to which the point belongs it’s discarded.

After this process we only have point clusters that represent objects on top of the table and the point cloud of the table top part. This are the environment’s components that we want to be perceived as obstacles. The only part left is taking this point clusters and transform them into representations that the planning pipeline can perceive as obstacles. We chose to represent them by their Oriented Bounding Boxes (OBB). We use our function *bounding box* with each of these clusters. This function computes the OBB through the moment of inertia estimation of the point cloud implemented in the PCL library.

These bounding boxes are then sent through a created topic called `/obj_recognition/collision_objects` further down into our approach pipeline. This node we just described is continuously running, performing the actions described above in a loop until it stops receiving an environment point cloud.

In Figure 4.1, a flowchart representing the above process can be seen with all the relevant processes, ROS nodes and ROS topics.

![Figure 4.1: Implementation flowchart for obstacle detection.](image)

### 4.2.2 Collision-free trajectory

To first be able to calculate a trajectory without collisions, we first need to pass to MoveIt the parts of the environment that belong to obstacles. To do this we launch a node called `work_scene`. This custom node subscribes to the topic `/obj_recognition/collision_objects` mentioned above. This node first removes all obstacles that might have been present in the planning scene from a previous iteration. It then extracts pose and dimensions from the bounding boxes received through that topic, converts them into a MoveIt obstacle and then adds all of the obstacles into the MoveIt planning scene by publishing to the topic...
planning_scene, a MoveIt topic created for this purpose. We performed tests using RRT and RRT* but, unfortunately, we were not able to use Informed RRT*, since it’s MoveIt implementation as some know issues. We can now calculate the trajectory through the environment, while avoiding collisions with the obstacles present in it. To do this we launch our node called trajectory_plan. First MoveIt needs to know what is the planning group to which it will calculate the trajectories. This means indicating which of the robot’s arms (or both) is going to be performing the calculated path. After setting this information, the planning framework needs to know what is the desired end effector final pose, this is, the position and orientation in which we want the end effector to be at the end of the trajectory (it takes as the initial pose the current end effector pose). Afterwards, the planner to be used is set by it’s OMPL identifier. We can now calculate the trajectory. MoveIt checks the planning scene for the obstacles and executes the chosen planner algorithm and computes the desired collision-free trajectory. This trajectory can then be executed by the robot if the user wishes to.

Instead of using this node we created (trajectory_plan), the user can also set the robot’s end effector final pose and planner through the MoveIt Rviz plugin, which allows for immediate visualization and execution of the calculated trajectory.

In Figure 4.2, a flowchart representing the above process can be seen with all the relevant processes, ROS nodes and ROS topics.

![Flowchart](image)

**Figure 4.2:** Flowchart with the process for passing the detected obstacles to MoveIt and for path planning and execution.

We performed tests using RRT and RRT* but, unfortunately, we were not able to use Informed RRT*, since it’s MoveIt implementation as some know issues.
4.2.3 Visual Servoing

After following, in open-loop, the trajectory calculated through MoveIt that avoids the obstacles detected through the method above explained, we now want to correct the possible end effector pose errors in relation to its current pose and the pose we desire it to have at the end of the trajectory. As mentioned before we do this through Eye-in-Hand Image Based Visual Servoing.

This node executes, looping through the above process, while it receives current camera images or the error in the visual servoing task is above a defined threshold. This is done because subscribing directly to the camera’s topic wouldn’t allow our application to distinguish the current from the reference image. The reference image node is called when we position the end effector in its desired final pose and only runs once, since this reference image doesn’t change. The current image node is continuously running, because the end effector will move and thus changing the camera’s image, and it’s activated when the end effector finishes executing its trajectory.

The node responsible for the implementation of our image feature detection and visual servoing is called `SURF_match`. When this node receives one of the above images it converts them to a OpenCV format, the library we used for image manipulation and feature detection. After receiving the images, we perform SURF feature extraction on both of them and then we do k-nearest neighbors matching between the features on the two images.

In these matches there are two types of matches that we don’t want to consider. The first type of these matches is the wrong matches and to remove them we perform Lowe’s Ratio Test, which takes a feature and compares the distance of its two best matches in the second image. If the ratio between these two matches is above a certain threshold then the match is discarded. But the second type of features we want to eliminate are matches with the robot’s end effector, which appears in the camera image. To remove them we created an end effector mask (the end effector is always on the same positions on the image) and any match in the mask area is discarded.

After this we initialize our visual servoing task as an IBVS Eye-in-Hand task in which the interaction matrix will be calculated in relation to the desired values of the features (as mentioned before, so we don’t have varying feature depth). This is done by setting some ViSP variables. Then we need to add the features to the ViSP visual servoing task, but the SURF detection process gives the features coordinates in pixels. To convert the pixel coordinates to x and y we take the camera’s intrinsic parameters from the topic `/cameras/(left/right)_hand_camera/camera_info`. After doing this conversion we can add the features to the visual servoing task. Finally, ViSP calculates the end effector velocity that will minimize visual feature error in the camera’s frame. Since in each iteration of this process we can have a varying number of visual features, we need to “kill” the ViSP task in each iteration since this is the only way to remove features from it.

Now we need to translate the camera frame velocities to joint velocities to give to the robot’s con-
trollers. To do this we need the robot’s end effector jacobian matrix. To get this matrix we use MoveIt’s kinematic state interface, which first needs the current robot’s joint positions which we take from the topic /robot/joint_states. After getting the jacobian we calculate it’s Moore-Penrose Pseudoinverse through the SVD method.

However, the jacobian matrix is defined in relation to the robot’s base frame and the velocity is in the camera’s frame. We then need to rotate the velocity vector calculated by ViSP in a way that it is aligned to the base frame of the robot. By using the TF trees (the robot’s camera frame is in the tree) we can get the rotation matrix between these two frames and by multiplying this matrix to the velocity column vector we get the camera’s desired velocity expressed in the robot’s frame.

By multiplying the pseudoinverse of the jacobian with the transformed velocity matrix we get the joint velocities to be applied to the arm. This would be the final step if the Gazebo simulator was able to handle joint velocities given directly to the robot, but the simulator as some unexpected behavior while doing this. Therefore we need to give to the robot joint positions instead of joint velocities. We take the current joint positions used before and add to them the intend joint velocities with a certain time duration. This will give the joint positions as if those joint velocities were applied to the robot during a certain time. This new joint positions are then sent to the robot through the topic /robot/limb/(left/right)/joint_command.

This node executes, looping through the above process, while it receives current camera images or the error in the visual servoing task is above a defined threshold.

In Figure 4.3, a flowchart representing the above process can be seen with all the relevant processes, ROS nodes and ROS topics.

![Flowchart of the real-time feature calculation and visual servoing.](image-url)
## 5 Experimental Results

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Experimental Setup</td>
<td>33</td>
</tr>
<tr>
<td>5.2 Detection of obstacles</td>
<td>33</td>
</tr>
<tr>
<td>5.3 Collision-free trajectory</td>
<td>35</td>
</tr>
<tr>
<td>5.4 Visual Servoing</td>
<td>36</td>
</tr>
</tbody>
</table>
5.1 Experimental Setup

In this chapter we will present the results of the implementation of the proposed approach, as described in the previous chapters. But first we will define our experimental setup, in order to better understand the context in which the experiments were done.

The experiments done were conducted in a simulated environment in the Gazebo simulator. Our world is comprised of a 3D sensor in a pedestal to give it an appropriate height, a table with some objects on it and a robot.

The robot we used in our experiments was the Baxter Research Robot. This robot has two arms with seven degrees of freedom, with a camera in the wrist of each one. At the end of its arms there is a simple, two fingered gripper. On Figure 5.1 we can see an example of this simulated world in the Gazebo simulator.

![Figure 5.1: Example Gazebo scene similar to the ones used for testing our implementation of the proposed approach.](image)

5.2 Detection of obstacles

In this section we will explain and present the results of our implementation of the process to detect obstacles present on top of a table, as it was explained in the two previous chapters. In Figure 5.2 we can see the full point cloud as it is perceived by the depth sensor (without any filtering of its components). This image shows one of the vulnerabilities of the approach used: the positioning of the camera. In this case the camera was not properly placed and so the top part of the can (outlined in red) is not completely
represented which will lead to an incorrect bounding box (as can be seen in the left image). With proper adjustment of the camera’s tilt angle the whole scene will be correctly represented (as can be seen in the right image).

![Figure 5.2: (a)- Bad object representation due to unadjusted camera pose. (b)- Good object representation due to proper adjustment of the camera.](image1)

However, in the above figure we can still see points that represent parts of the arms of the robot (the black parts after the edge of the table). As was explained before, this constitutes a problem since our application will perceive them as obstacles and when we try to plan a trajectory for the end effector a collision will be detected at the starting state. In Figure 5.3 is shown the filtered environment containing only the objects of interest. The blue lines represent the lines that go from one consecutive joint of the arm to another and which are vital to removing the clusters representing parts of the arm.

![Figure 5.3: Results of environment filtering and arm removal from the point cloud.](image2)

In Figure 5.4 we can see the final result of the process of filtering the environment and calculation of the bounding boxes of the desired object clusters (and table’s top). As can be observed the bounding boxes are coherent with the segmented point cloud clusters presented above. However, another vulner-
ability of the generalization of object shape to bounding box is seen. In objects like the can or others, like bottle for example, the bounding box’s volume contains parts that do not belong to the object. Although this does not harm the trajectory calculation algorithm, it may lead to sub-optimal paths. This is because parts of the environment are being represented as occupied space when in fact they are free and can be navigated through.

![Figure 5.4: Bounding boxes representing the obstacles in the scene.](image)

### 5.3 Collision-free trajectory

Since the RRT path-planning implementations (RRT and RRT*) used were the ones "out of the box" from MoveIt, their behavior was predictable and without unexpected results. In addition to using the base RRT implementation and RRT*, we also used RRT Connect [25], which improves slightly on the original RRT implementation talked about before. This implementation grows two search trees, one from the goal state and one from the start state. When the trees connect, a path has been found.

The experiments done with the RRT’s were done to verify the functioning of the process of translating the bounding boxes to MoveIt obstacles. In our experiments we found that both base RRT and RRT Connect produced longer paths with unnecessary movement because they don’t optimize path length. RRT* produced a much shorter path. This can be seen in Figure 5.5.

This added path optimization comes with an increased computational cost. In our tests we found that, on average, base RRT took 2.3 times more time to plan than RRT Connect and RRT* took 100 times more. RRT and RRT Connect usually took tenths of a second to plan, while RRT* took seconds (for the same end effector start and initial pose).
Figure 5.5: (a)- RRT calculated path. (b)- RRT* calculated path.

5.4 Visual Servoing

The first step in testing our implementation of our visual servoing controller with real-time calculated visual features was to test which feature extracting method would be used. We ran some tests with the SURF and SIFT algorithms explained in previous chapters in order to compare them and to determine which would be the most appropriate to our task.

In our test we noticed that SIFT keypoints tend to be more grouped up in an area of the image, for example, around the contours of letters or shapes. SURF keypoints tend to be more spread out around the object. This is better from a visual servoing standpoint since when the features are more far apart they provide better information to the servoing since the change in feature position on features very close to each other is very similar, which doesn’t happen when the features are more separated.

In Figure 5.6 we have two images representing the feature matching process of SIFT and SURF where we can see in the SIFT image the features tend to be more clustered (for example around the letter E in the red part of the can).

SURF is also, according to its creators, to be more quick in its feature detection due to the use of integral images. This makes it more appropriate for real-time estimation of features.

After choosing the most appropriate method for feature extraction and matching, we ran some tests in order to determine the best value for the Lowe’s Ratio Test. As was explained before, this test is the method with which the incorrect feature matches are removed. This is done by taking a keypoint from the reference image and finding the two best matches on the current image. If the two matches are too close (in terms of descriptor distance) to each other keypoints are both images are considered a bad
match and are discarded.

We conducted a series of tests, each test with a different value for the Lowe's Ratio. In Figure 5.7 are presented images of the results of the previously mentioned tests with the ration set as 1 (all matches without filtering anything), 0.7, 0.5 and 0.3.

As was to be expected, the first image has a lot of incorrect matches, since no kind of match filtering is done. On the second image, although the ratio is set to a value below the one proposed in [17] of 0.8 (which in their case eliminated 90% of incorrect matches), there are still some incorrect matches. On the third and fourth images, all matches are valid ones, although in the last image there are very few matches since the filtering is too "harsh".

To test the impact of having incorrect matches in the visual servoing, we tested the visual servoing with the ratio threshold having various values. To do these tests we first put the robot's arm in the desired final position and then moved it to another pose and ran our visual servoing program. In Figure 5.8 we can see a graph of the results of these experiments. The error in the graph is normalized to the number of features since in each iteration we can have a different number of features matches and with changing the ratio threshold also changes the number of total feature matches as was seen by the previous image. This error is calculated by the sum of the squared errors between current and desired features (since the features are defined by their coordinates it measures the difference in image coordinates).

As was expected, with the ratio set to 1, meaning no filtering is done, the system became unstable. With lower values of the ratio the system eventually trended towards the minimization of the error (even with ratio values where there are still incorrect matches). Another important conclusion is when we are stricter with the features chosen, the lower the error is and the quicker the system trends towards the desired result.
One curious fact that can be observed in the graph is that the normalized error starts quite low, rises up and, when it does not become unstable, goes back down. This is because, at the start of the task the end effector is still a bit far away from the object and therefore the number of features is not that high. When the end effector starts going towards the object more and more quality feature matches are found and the normalization of the error is not enough to keep the error from rising, although the average individual feature error is lower since the end effector is moving towards the desired pose. The error starts lowering again because the the number of features stabilizes and the end effector keeps moving towards the final desired pose.

To finalize, the end effector presents a stuttering motion. This happens because, during development we had a problem with the Gazebo simulator and its handling of direct joint velocity control. In some joints, giving a velocity to that joint would not make the robot the desired and expected motion. Because of this we had to instead use joint position control. To do this we approximated the joint velocity by its position discretisation over time. This leads to a stuttering motion towards the target.

In Figure 5.9 we can see three frames taken during the servoing task, one representing the start and end desired states, another from the middle of the task and finally one nearing the end of the task.
Figure 5.8: Normalised error of the visual servoing features with the variation of the ratio threshold.

(a)

(b)

(c)

Figure 5.9: Frames at the beginning (a), middle (b) and end (c) of the visual servoing task.
6

Conclusions

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Conclusions</td>
<td>41</td>
</tr>
<tr>
<td>6.2 Future Work</td>
<td>41</td>
</tr>
</tbody>
</table>
6.1 Conclusions

In this work we proposed doing a path planning approach towards a given end pose that doesn’t require an hand-crafted 3D model a priori. The path is followed in open-loop and at the end of the motion of the arm, the errors from this execution are be fixed by using Eye-in-Hand Image Based Visual Servoing with real-time calculated features in order to close the control loop around image information.

In this approach we first started by taking the 3D information given by an exterior depth sensor overlooking the environment and filtered it until we got the parts that we wanted to be considered as obstacles to our task. Since the presence of the arms in the point cloud of the environment could lead to them being considered as obstacles, we devised a strategy to remove them from the representation of the environment, by way of its current joint positions.

We then calculated a trajectory using sample-based path planning algorithms in order to validate our representation of the obstacles present in the work space. The calculated trajectory was then followed in open loop.

After the execution of the calculated path, the final pose errors were corrected through way of visual information. This information was collected from the camera attached to the end effector of the robot. By using a reference image and the current image, features were extracted and matched between those pictures. The valid matches were then translated into visual servoing features and were used to drive the end effector to the correct final pose.

With this approach we implemented path-planning without the existence of a previous model of the work space. This approach is therefore much more adaptable to new work space configurations. We also implemented a markerless image based visual servoing strategy with arbitrary features, calculated in real-time. This provides a much more flexible approach since there is no need for added markers in the environment since the features are notable points in it.

All the code developed for this thesis can be seen in the GitHub repository in https://github.com/MGRNascimento/Tese. With this thesis we hope to spawn further research in both model-free path planning and markerless visual servoing.

6.2 Future Work

In future work we would like to use a more complex way to represent the obstacles and to mitigate the reliance on correct 3D sensor placement. This could be done by using point cloud reconstruction methods in order to get a full point cloud from a partial one. As for the representation of the obstacles, superquadric segmentation [26] into smaller, parametric polygons could be a valuable addition.

An integration of visual servoing during trajectory execution would also be important. Although occlusions will always be a problem when it comes to cluttered environments and processes reliant on visual
features, there are ways to mitigate this problem. One way is, for example, to implement a switching scheme which keeps track of feature visibility. When not enough features are present, the system operates in open-loop but, as soon those features appear again in the camera image, the system switches back to visual servoing. Another important addition would be the incorporation of a depth estimation method. In this work we used a roughly estimated desired feature depth since, without any model, is very difficult doing monocular depth estimation. Although it does not influence the task’s success rate, it influences the speed of convergence of the visual servoing controller.
Bibliography


