Visual Servoing with Collision Avoidance using Rapidly-exploring Random Trees

Miguel Nascimento
miguelgrnascimento@tecnico.ulisboa.pt
Instituto Superior Técnico, Lisboa, Portugal
December 2019

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: Visual Servoing, 3D Vision, Collision Avoidance, Rapidly-exploring Random Trees.

1. Introduction

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.

Robots are extremely complex machines, with long kinematics chains which makes them hard to calibrate. Therefore, when executing using 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 often 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 standardized 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.

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 usage 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 re-
quire the offline acquisition of the environment’s 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. This 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.

The rest of this document is organized as follows: in Section 2 a review of the theoretical background of visual servoing and RRT’s is given. In Section 3 the relevant related work to this thesis is briefly presented. In Section 4 an explanation of the followed methodology and of the theory behind some of the algorithms used is presented. In Section 5 implementation details of the proposed solution are given. In Section 6 the results of the tests done to validate our implementation are presented and explained. Finally, in Section 7 the conclusions we took from the developed work are given and the future work to give continuation to this thesis is proposed.

2. Background

2.1. Visual Servoing

Visual Servoing is a technique that uses visual feedback gathered from a vision sensor (a camera, for example) in order to control the motion of a robot. The aim of controllers based on visual servoing is to minimize the error function $e(t)$

$$e(t) = s(m(t), a) - s^*, \quad (1)$$

where $s$ and $s^*$ are the current and desired visual feature values, respectively, $m(t)$ is a set of image measurements (for example, image coordinates for interest points) and $a$ is a set of parameters that gives additional knowledge about the system (e.g., camera intrinsic parameters).

This general formulation encompasses many different approaches for using this type of feedback control. This equation expresses the minimization of the difference between the current value of the visual features selected ($s(m(t), a)$), like coordinates of selected image points, and the desired final value ($s^*$).

The visual feature vector $s$ is chosen according to the type of visual servoing that is being used. There are two main types of visual servoing: Image Based Visual Servoing (IBVS) and Position Based Visual Servoing (PBVS). IBVS uses features present directly in the camera’s image while PBVS utilizes a set of 3D parameters calculated from image measurements. Later on in this section we will go on further detail about these two types of controllers.

Once the feature vector is chosen a controller must be designed. A commonly used one is the velocity controller

$$v_c = -\lambda \hat{L} e, \quad (2)$$

where $v_c$ is the velocity matrix on the camera frame, $\lambda$ is for exponential decrease of the error and $\hat{L}$ is the estimation of the pseudoinverse of $L_c = L_{ss}$. On [2] a more detailed explanation is provided for the formulation of this controller.

The way Image Based Visual Servoing (IBVS) works can be easily pictured has taking a set of image points and “moving” them to the final desired position on the 2D camera image, as can be seen in Figure 1 below.

![Figure 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].](image)

Traditional Image Based control schemes use the coordinates on the image plane of a set of points 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 $L_{sx}$, as can be seen below. $p$ is a point in the camera’s image defined by its $x$ and $y$ coordinates on the image frame. The matrix $L_{sx}$ can be defined in relation to the features in the current camera image or the desired camera image.

$$\dot{p} = L_{sx} v_c, \quad p = [x \ y]^T \quad (3)$$
Since IBVS is solely reliant on 2D camera image information some problems may arise from this. This method is prone to be attracted to local minima of the error function or to singularities of the interaction matrix. This is most likely to happen when the start and end poses are far apart. IBVS controllers are often associated to Eye-in-Hand robot configurations (hand and gripper are attached) since the motion of the end effector has a direct correlation to the motion of image points. 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.

While IBVS is based purely on 2D information, Position Based Visual Servoing (PBVS) is a 3D approach. It uses the relative pose of the camera to some reference coordinate frame, calculating this pose using image measurements, camera intrinsic parameters and the 3D model of the observed object. The features to be used to control the servoing can be selected in different ways, leading to different PBVS control schemes, leading also to different camera motion and trajectory. These features can be defined in relation to the current or desired camera frames.

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, which can be aggravated when 3D model errors also exist. 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.

2.2. 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 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.

In Figure 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, $x_{start}$. To extend the tree a random state vector $x_{rand}$ is calculated and the nearest state vector $x_{near}$, in the state space of the considered environment, to $x_{rand}$ is found. $x_{near}$ 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 $x_{rand}$ will be calculated. This process is repeated until $x_{goal}$ is reached.

This describes the base formulation of this algorithm but there are different implementations. One of those is RRT* [3]. 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* [4] 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 3 we can see a comparison between searched space states when using RRT* and Informed RRT*.

3. Related work
In this section we review the current works in the field of visual servoing and the application of RRTs in this subject.

In [5] the authors define three kinds of approaches for motion planning in robotic grasping and manipulation. First, Sense-plan-act is a strong
modular approach that divides the servoing task into smaller sub problems (separates environment sensing, path planning and trajectory execution) but does not adapt well to changes in the environment, Locally Reactive Control, where only the local geometry near the end effector is considered, and Reactive Planning, which is a combination of the two approaches above, but it has few implementations on robots with a high number of degrees of freedom. Our work fits between sense-plan-act and Locally Reactive Control since it uses components from both elements, namely by doing a representation of the environment and computing a collision-free path and then using a locally reactive controller to correct end effector pose errors (visual servoing).

Current applications of the RRT algorithm to the field of Visual Servoing are quite limited since they require a pre-made model of the environment. On [6] we can see one such implementation. This works uses an IBVS controller with an Eye-in-Hand robot configuration. A model of the environment is fed to the RRT algorithm. The path calculated is then followed through visual servoing by tracking the visual feature’s trajectory in the camera image.

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. However, most works tend to use markers in order to quickly calculate image features. These presents satisfactory results but introduces a degree of artificiality to the environment and reduces the adaptability of the systems. In [7] SURF features [8] are used with an IBVS controller. By doing feature matching on the reference and current image, they select a region of interest which will be tracked throughout the motion of the camera. This is done to reduce the computation time in finding and comparing features. They use geometrical measurements taken from this region of interest as the visual features used in the controller.

In [9] it is presented 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 in order to mitigate the errors in the inverse kinematics of the robot. This is done by adding the offset between real joint angles and measured joint angles. By doing this, the robot’s internal model is continuously being fixed to its correct state. The offset value is calculated comparing the captured images and images generated in simulation.

4. Methodology
In this section we will explain the methodology used to develop the solution for this thesis. In Figure 4 we can see the pipeline for the solution with its main components.

Through the 3D information provided by a RGB-D camera we obtain a point cloud representation of the environment. On this representation we identify the obstacles present in the environment and then calculate a collision-free trajectory using RRT. We then execute this trajectory in open-loop until the end effector reaches the vicinity of the desired pose. Error correction using IBVS is then performed using the visual information from the 2D camera attached to the robot’s arm. We will go into further detail about each of the main components in this section.

4.1. Obstacle Detection
In order to detect the obstacles present in the environment where the robot will operate we first need to obtain a representation of it. We do this by taking the point cloud information from a RGB-D camera overlooking the environment.

Having this representation we then need to ascertain what constitutes an obstacle. We take as an assumption that the working scene is a table with some mundane objects on top of it. With this assumption we can perform tabletop segmentation to identify the point clusters that represent the obstacles in the point cloud representation of the environment. The extracted table point cloud is also considered an obstacle to our task.

4.1.1 Filtering the environment
To perform tabletop segmentation we first use the Random Sample Consensus (RANSAC) [10] algorithm to identify the dominant plane in the point cloud, in this case being the plane of the table’s top part. This algorithm works by detecting the inliers in a set of data that fit a certain model. It functions by selecting a randomly some data and calculating the model that fits said data. It then compares the rest of the data to that model. The algorithm stops when enough elements of the rest of the data set fit the proposed model. The model estimated by RANSAC in our approach is a planar model.

After removing all inlier points belonging to the table, the point cloud is now constituted solely by the objects and parts of the robot that happen to be in view of the camera. However, we don’t want...
the arms of the robot to appear in the point cloud. This is because if they were seen as obstacles by our program then the arms wouldn’t move since the system would think a collision is occurring. To remove the arms we use the robot’s internal model. We calculate the lines between each consecutive joint in the arms of the robot and, if a point is within a given radius of those lines, it is discarded.

4.1.2 Grouping the points into clusters

Now we have a completely filtered point cloud with only the points belonging to the objects. We now need to group up the points into clusters that form compact objects, one cluster for each obstacle. This is done by using k-d trees and nearest neighbor search. These are a type of binary trees that separates points according to their position. Representing the point cloud in this way makes it easier computationally to perform a nearest neighbor search. If many points are in the neighborhood of one another, they belong to the same cluster.

When all points have been assigned a cluster we have a representation of each individual object in the original point cloud.

4.2. Collision Avoidance

To calculate a collision-free trajectory we use algorithms already available in software frameworks for motion planning.

These trajectories are planned in 3D space. The planners build the trajectory through a series of pose waypoints and, 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 work space.

In the context of RRT’s 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.

After the trajectory is calculated, it is followed in open-loop, leading the end-effector to a vicinity of the target pose.

4.3. Visual Servoing

Once we reach a vicinity of the target configuration by following the open-loop trajectory, we can now correct the errors in the final pose of the robot. To do this we use an image based visual servoing controller that uses real-time calculated features in order to have a model-free correction of the errors. To do the feature calculation we compared two different types of features to represent the target object: SIFT[11][12] and SURF[8].

4.3.1 Visual Features

Scale-invariant feature transform (SIFT) [11][12] is a feature detection algorithm that finds and describes local features in an image. These features are scale-invariant and are resistant to photometric changes like illumination and contrast. They are often used to find specific objects in cluttered environments. The algorithm does this by matching features found in a training image and the ones found in test images. SIFT works by detecting keypoints through Difference of Gaussians (DoG) images. These images are calculated by making the difference of two Gaussian images at different scales. Keypoints are found at the local maxima and minima of the DoG images. After estimating the keypoint position, a descriptor is assigned to each one of them. The orientation descriptor is built through local image gradient directions. Another descriptor is computed by comparing gradient magnitude at image points around the keypoint.

Speeded Up Robust Features (SURF) [8] is also a local feature detector and was inspired by SIFT. SURF uses integral images for a better computational performance. 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 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. The dominant orientation is calculated by summing the responses within a defined sliding window. As for the descriptor a similar process to the SIFT descriptor is done but instead of using gradient magnitude and orientation, SURF uses the response to Haar wavelet.

4.3.2 Visual Servoing Controller

From the visual features in the camera’s image, we need to define the visual servoing control law.

To define the desired final position of the end effector, we position the robot in that exact position in a calibration phase, and we take a reference image and store it for later use. Then, in run time, we compare the features extracted in the current camera image to the features in the reference image collected offline. We then match features in both images by using k-nearest neighbors. This matching method finds many matches between the keypoints on the reference and current image, but some of them are incorrect matches. To filter out these bad matches, Lowe’s Ratio Test [12] is used. This method takes a keypoint from the reference image and its two best matches on the current image. If these matches have a similar distance to the keypoint on the reference image, they are dis-
carded. This is done because the matching was not “unique” enough, it was close to being matched to another image point.

With a set of dependable keypoint pairings, we can now use them as the reference and current image visual features to be used in the visual servoing controller. We used a velocity controller similar to the one in (2). As for the estimation of the interaction matrix, we chose to define it in relation to the desired image visual features.

4.3.3 Controlling the robot

With the method described above, the controller calculates a velocity to be applied to the camera. We now have to make the robot apply this velocity to its end effector. Therefore, we need to translate this velocity to a set of joint velocities to be applied to the robot’s arm.

In manipulators with ix degrees of freedom this process is relatively simple by directly using the robot’s Jacobian matrix. This matrix translates the velocity of a given joint into a certain Cartesian space change, as can be seen in (4), in which \( \dot{q} \) represents 6 DoF velocity, \( J \) is the robot’s Jacobian matrix and \( \dot{q} \) is the vector of joint velocities.

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

(4)

Therefore, for a non-redundant manipulator, we only need to invert the Jacobian matrix and multiply it by the Cartesian velocity vector.

When using a manipulator with a greater or lesser number of DoF than 6 this is a bit more complicated since the Jacobian matrix is non-invertible and there are infinite possible solutions. To solve this we use the approach in [13]. The Moore-Penrose pseudo-inverse of the Jacobian, \( J^+ \), is used instead of the normal matrix inverse. The joint velocities are then calculated by

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

(5)

With this, we now have a joint velocities vector that will move the robot’s end effector to the desired final pose, thus correcting any error there was in its final pose after the execution of the collision-free trajectory.

5. Implementation

In this section we will go through some details on the implementation of the proposed solution and of the software used.

5.1. Software Framework

We used ROS [14] for communication between our application and the robot, and for communication between our software. ROS 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. It is built upon two main concepts: nodes and topics. Nodes are where the user’s code is run. Nodes can communicate with each other by subscribing and publishing to topics.

To simulate the test environment and the robot’s response to our solution we used the Gazebo simulator [15]. Interaction between our solution and Gazebo is done through ROS topics, allowing for control of the robot’s model in the simulator.

In order to perform collision-free path planning we use the MoveIt motion planning framework [16]. Movelt runs on top of ROS and its available through a ROS package and incorporates many internal and external packages. Movelt performs planning through an external library called Open Motion Planning Library (OMPL) [17], which is a motion planning library with several planners implementations but without the notion of a robot and so Movelt 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 Movelt 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.

Finally, for the implementation of the visual servoing controller, we used ViSP [18]. ViSP 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.

5.2. Software Architecture

In Figure 5 we can see the implementation schematic for the detection of obstacles. We take a point cloud from the 3D camera and transform it to world coordinates. Then the operations described in 4 are done on this point cloud in order to get the point clusters of the objects. The bounding boxes of the point clouds are then calculated.

In Figure 6 the process to transform the bounding boxes to Movelt obstacles is shown. All obstacles are removed from the planning scene and the bounding box’s pose and dimensions are translated into an obstacle and published into the planning scene. To calculate and execute the trajectory, Movelt receives a final pose and a trajectory planner. It then calculates the trajectory and executes it through interaction with the robot’s controllers.

In Figure 7 the feature extraction and visual servoing implementation is explained. From the robot’s end effector camera’s the reference and
current images are extracted. Features are extracted and filtered as explained before in section 4 and the features are added to the visual servoing task and the camera velocity is calculated. The robot Jacobian is taken from MoveIt, but to do this the joint positions of the robot need to be updated. Joint velocities are then calculated and translated into change of the robot’s current joint positions. This new robot joint positions are sent directly to the robot for execution.

6. Experiments and results

6.1. Experimental Setup

In this section we will present the results of the implementation of the proposed approach, as described in the previous sections. 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 realised 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.

6.2. 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. In Figure 8 we can see the full point cloud as it is perceived by the depth sensor (without any filtration of its components).

However, in this figure we can still see points that represent parts of the arms of the robot (the black parts after the edge of the table). By using the process explained previously we removed the arms of the robot.
In Figure 9 we can see the final result of the process of filtering the environment and calculation of the bounding boxes of the desired object clusters. The bounding boxes are coherent with the segmented point cloud clusters presented above. In the figure the obstacles and table can be seen represented through their bounding boxes, in addition to a bounding box outside of the table that belongs to the robot’s torso. There is no harm in this being considered an obstacle since it’s a non-moving part of the robot.

Figure 9: Bounding boxes representing the obstacles in the scene.

6.3. Collision-free trajectory

The RRT path-planning implementations (RRT and RRT*) used were the ones “out of the box” from MoveIt and their behavior was predictable and without unexpected results. In addition to using the base RRT implementation and RRT*, we also used RRT Connect [19], which improves slightly on the original RRT implementation talked about before by growing two trees instead of just one. One search tree starts from the initial pose and the other starts from the end pose. When they connect a valid 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 10. In this figure it is presented three frames, one from the beginning, one from the middle and one from the end of a trajectory calculated using RRT and RRT*. On the second frame it can be seen that RRT produces an exaggerated motion while RRT* produces an optimized trajectory.

This added path optimization comes with an increased computational cost. In our experiments we found that RRT* takes two orders of magnitude more than planning with RRT and RRT Connect (RRT* took seconds while RRT and RRT Connect took tenths of seconds).

Figure 10: (a)- RRT calculated path. (b)- RRT* calculated path.

6.4. Visual Servoing

To test our visual servoing approach, we first started by comparing feature detection results by using SIFT and SURF and choosing which own is better for our case.

In our test we noticed that SIFT keypoints tend to be more grouped up in an area of the image. 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.

In Figure 11 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.

Figure 11: (a)- Feature matching with SIFT. (b)- Feature matching with SURF. Both images were taken with a Lowe’s Ratio at 0.5.

After choosing the most appropriate method for feature extraction and matching, we ran some tests in order do determine the best value for the Lowe’s Ratio Test.

In Figure 12 are presented images of the results of said tests with the ratio set as 1 (all matches without filtering), 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 filtration is done. On the second image, although the ratio is set to a value below the one proposed in [12] 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.

Figure 12: (a)- Lowe’s Ratio 0.3. (b)- Lowe’s Ratio 0.5.
To test the impact of having incorrect matches in the visual servoing, we tested the visual servoing with the ratio threshold having various values. In Figure 13 we can see a graph of the results of these experiments. The error in the graph is normalized to the number of features because the number of features varies during the servoing. 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 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. This could be avoided by choosing a fixed number of the best feature matches.

In Figure 14 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.

7. 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 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 workspace. 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 arbitrary features, calculated in real-time. This provides a much more “real” approach since there is no need for added markers in the environment.

All the code developed for this thesis can be seen in the GitHub repository in https://github.com/MGRNascimento/Tese.

7.1. 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 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 as 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.

References
[16] Ioan A. ¢ ucan and Sachin Chitta. MoveIt!