2D Visual Servoing meets Rapidly-exploring Random Trees for collision avoidance

Miguel Nascimento[1], Pedro Vicente[1], and Alexandre Bernardino[1]

***Abstract*—Visual Servoing is a well-known subject in robotics. However, there are still some challenges on the visual control of robots for applications in human environments. In this article, 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 segmenting the environment and detecting the obstacles in the scene. 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 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 segmentation, trajectory calculation and error correction through visual servoing) of our solution. Furthermore, our solution was implemented in ROS using the Baxter Research Robot.**

Keywords: 2D Visual Servoing, Depth Camera, Collision Avoidance, Rapidly-exploring Random Trees.

I. Introduction

Nowadays it is expected for robots to operate on human environments. However, these environments are usually complex. Apart from the target object, other objects are cluttering the scene. This turns out to be hard for a robot to execute the task since it does not know how to avoid the obstacles while trying to reach its goal.

Robots are extremely complex machines, with long kinematics chains which makes them hard to calibrate. Therefore, when executing a reaching task using an 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 (VS) has been employed often as the method to solve the problem of open-loop controllers by closing the control loop with visual feedback.

This work was partially supported by FCT with the LARSyS - FCT Plurianual funding 2020-2023 and the PhD grant PD/BD/135115/2017.

[1] M. Nascimento, P. Vicente and A. Bernardino are with the Institute for Systems and Robotics, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal. miguelgrnascimento@tecnico.ulisboa.pt {pvicente,alex}@isr.tecnico.ulisboa.pt

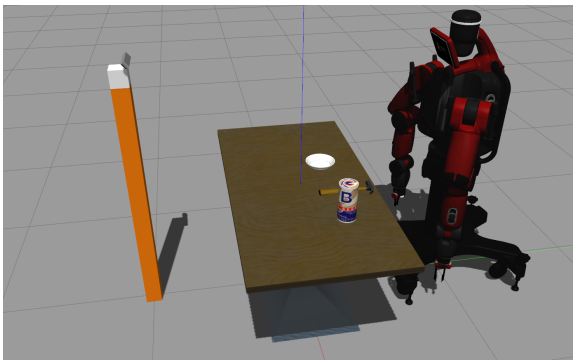


Fig. 1: Scene containing the Baxter Robot, a 3D camera and a table with objects

However, VS approaches do not have a standardised way to avoid obstacles in the environment. Therefore, there is a need to calculate a trajectory through the environment while avoiding obstacles. Following trajectories with VS in a cluttered environment can be very hard because of the high probability of occlusions.

The objectives for this work can be divided into two parts: i) the calculation and execution of a collision-free trajectory through a cluttered workspace and ii) 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 modelled workspaces. In order to build a more adaptable system we propose the usage of 3D information collected from a depth sensor over-viewing the environment in which the robotic end-effector operates. Through this information, the obstacles will be detected and sent to a path planning algorithm, along with a desired final pose, to calculate a collision-free trajectory. This will allow greater flexibility in which this type of algorithm can be applied to since it does not require 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 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, in particular, on tasks that require precision, like grasping and manipulation of objects. VS 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 feature values. In this work, to correct the end-effector final pose achieved from the open-loop execution of the trajectory, we will use a VS controller that uses real-time calculated 2D visual features to take the robot's end-effector from its current pose to the desired one.

II. Related work

Kappler *et al.* [2] define three kinds of approaches for motion planning in robotic grasping and manipulation. The *Sense-plan-act* (i)) is a strong modular approach that divides the servoing task into smaller subproblems (separates environment sensing, path planning and trajectory execution) but does not adapt well to changes in the environment, ii) *Locally Reactive Control*, where only the local geometry near the end effector is considered, and iii) *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 coarse representation of the environment and computing a collision-free path and then using a *locally reactive controller* to correct errors in the end-effector pose using visual servoing.

Current applications of the RRT algorithm to the field of Visual Servoing are quite limited since they require a premade model of the environment. Kazemi *et al.* use an Imaged-Based Visual Servoing (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.

Most works on Visual Servoing tend to use markers in order to quickly calculate image features. This presents satisfactory results but introduces a degree of artificiality to the environment and reduces the adaptability of the systems. In [3] SURF features ([4]) are used within an IBVS framework. 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 [5] it is presented an Eye-to-Hand Position-based Visual Servoing 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.

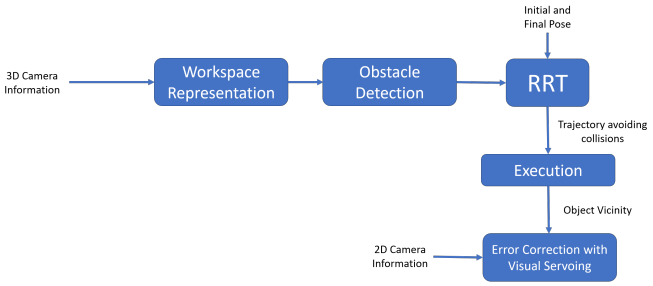


Fig. 2: Schematic for the approached used featuring its main components.

III. Methodology

The pipeline of our solution can be seen in Figure 2. Through the 3D information provided by an RBG-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.

A. 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 an 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.

1) Filtering the environment: To perform tabletop segmentation we first use the Random Sample Consensus (RANSAC) [6] 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 selects, randomly, part of the data and calculates the model that fits this 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.

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 neighbour 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 neighbour search. If many points are in the neighbourhood 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.

B. 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 way-points and, through inverse kinematics, an arm configuration that does not produce a collision with the rest of the environment is found. The pose way-points are calculated through sampling of the 3D workspace.

In the context of RRTs, 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.

C. Visual Servoing

Once we reach the 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. The target features are extracted based on SURF features [4].

1) Visual Features: Speeded Up Robust Features (SURF) [4] is a local feature detector and was inspired by SIFT [7]. However, 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 neighbourhood 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.

2) Visual Servoing Controller: The aim of controllers based on visual servoing is to minimise the error function $e(t)$

$$e(t) = s - s^*, \tag{1}$$

where s and $s*$ are the current and desired visual feature values, respectively.

To define the desired final position of the end effector, the robot is placed on the target position at a calibration phase, and the reference image is taken and stored. Then, in run time, we match the features extracted from the current camera image with the one from the reference image using k-nearest neighbours. To filter out some mismatches, we use Lowe's Ratio Test [7]. With a set of dependable keypoint pairings, we can now use them as visual features for the visual servoing controller. We used the velocity controller proposed by [8]:

$$v_c = -\lambda \widehat{L_e^+} e, \tag{2}$$

where v_c is the velocity matrix on the camera frame, λ is for exponential decrease of the error and $\widehat{L_e^+}$ is the estimation of the pseudoinverse of L_e. As for the estimation of the interaction matrix, we chose to define it in relation to the desired image visual features.

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.

We can exploit the robot's Jacobian matrix to do it. This matrix translates the velocity of a given joint into a certain Cartesian space change, as can be seen in (3), in which $\dot{v}$ represents 6 DoF velocity, J is the robot's Jacobian matrix and $\dot{q}$ is the vector of joint velocities.

$$\dot{v} = J\dot{q} \tag{3}$$

For robots with DoF different from 6, the Jacobian matrix is non-invertible and there are infinite possible solutions. To solve this, we use the approach in [9]. The Moore-Penrose pseudo-inverse of the Jacobian, J^+, is used instead of the normal matrix inverse. The joint velocities are then calculated by:

$$\dot{q} = J^+\dot{v}. \tag{4}$$

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.

IV. Implementation

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

A. Software Framework

We used ROS [10] for communication between our application and the robot, and for communication between our software. To simulate the test environment and the robot's response to our solution we used the Gazebo simulator [11]. 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 [12]. Moveit performs planning through an external library called Open Motion Planning Library (OMPL) [13], 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 through the Gazebo simulator.

Finally, for the implementation of the visual servoing controller, we used ViSP [14]. ViSP stands for Visual Servoing Platform and it is 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.

B. Software Architecture

In Figure 3 we can see the implementation schematic for the detection of obstacles. We take a point cloud from the 3D camera and transform it into world coordinates. Then the operations described in III 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 4 the process to transform the bounding boxes to MoveIt 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, MoveIt 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 5 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 III 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 a change of the robot's current joint positions. This new robot joint positions are sent directly to the robot for execution.

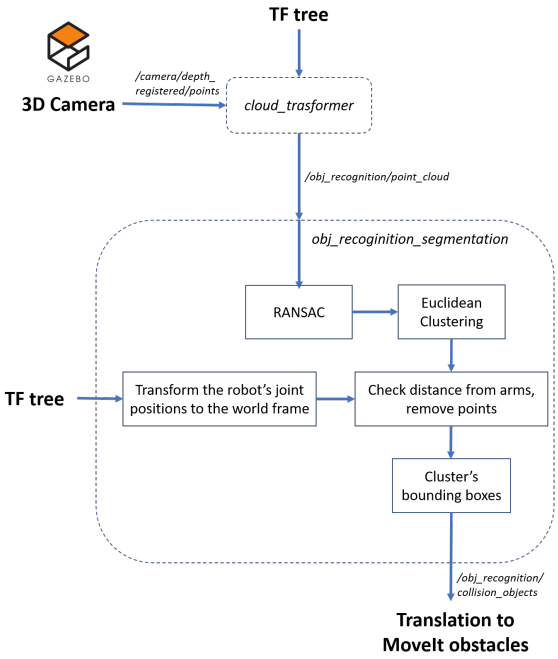


Fig. 3: Implementation flowchart for obstacle detection.

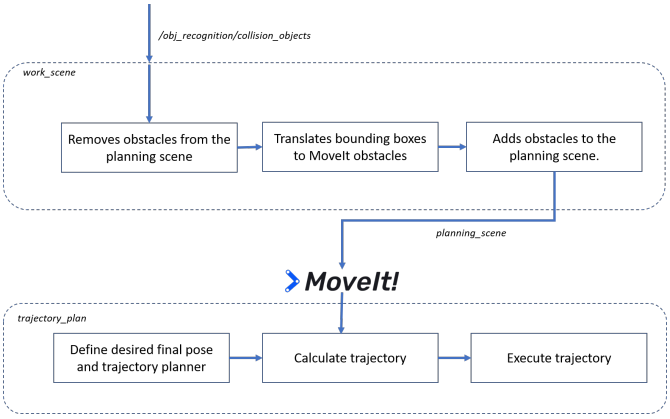


Fig. 4: Flowchart with the process for passing the detected obstacles to MoveIt and for path planning and execution.

V. Experiments and Results

A. Experimental Setup

The experiments were performed in a simulated environment using the Gazebo simulator. The scene is composed of a 3D sensor in a pedestal, a table with some objects on top and a robot - Baxter Research Robot. Baxter has two arms with seven degrees of freedom, with an RGB camera on the wrists. Moreover, it is equipped with a parallel gripper.

B. Detection of obstacles

In Figure 6 (a) we can see the full point cloud 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 6 (b) the final result of the filtering process and bounding box calculation can be seen. The bounding boxes are coherent with the segmented point cloud clusters presented

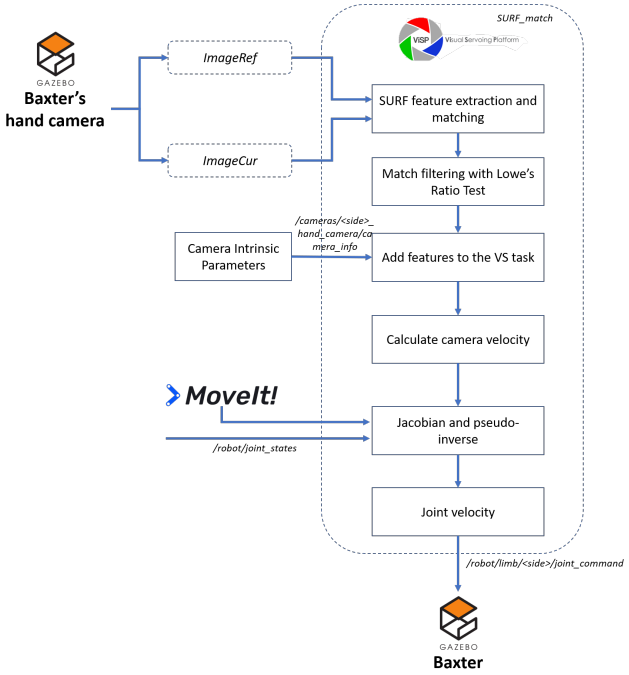
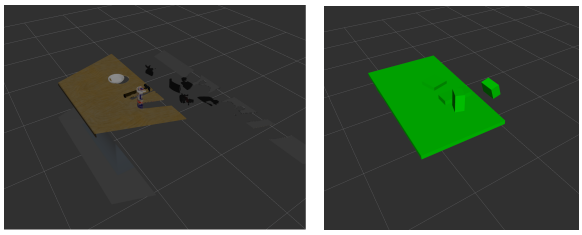


Fig. 5: Flowchart of the real-time feature calculation and visual servoing.



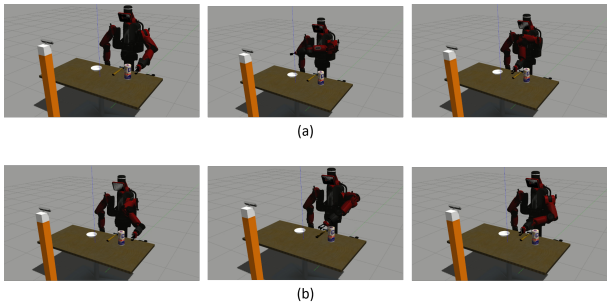
(a) (b)

Fig. 6: (a) Captured point cloud of the environment and (b) bounding boxes representing the obstacles in the scene.

above. Moreover, the obstacles are represented through their bounding boxes (objects, table and robot's torso).

C. Collision-free trajectory

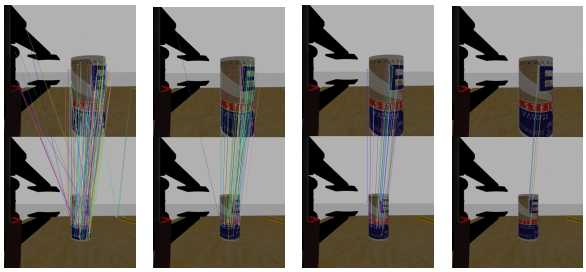
The RRT path-planning implementations (RRT and RRT*) used in this work were the ones provided within MoveIt package. In our experiments, we found that base RRT produces longer paths with unnecessary movement because it does not optimise path length. RRT* produced a much shorter path. This can be seen in Figure 7. Indeed, one can see 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 optimised trajectory. The path optimisation of RRT* comes with an increased computational cost. According to our experiments, RRT* takes two orders of magnitude more than planning with RRT (RRT* took seconds while RRT took tenths of seconds).



(a)

(b)

Fig. 7: (a) RRT calculated path. (b) RRT* calculated path.



(a) (b) (c) (d)

Fig. 8: Lowe's Ratio test: (a) 1. (b) 0.7. (c) 0.5. (d) 0.3.

D. Visual Servoing

In order to choose the best value for the Lowe's Ratio Test (*i.e.*, how to choose the number of features, we ran some test with several hypotheses. In Figure 8, the results of SURF features using the Lowe's Ratio Test are presented with ratios of 1 (all matches are used - 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 filtering is performed. On the second image, although the ratio is set to a value below the one proposed in [7] 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.

To test the impact of having incorrect matches in the control-loop, we tested the visual servoing with the ratio threshold having several values. In Figure 9 we can see a graph of the results of these experiments. The error in the graph is normalised 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 it was expected, with the ratio set to 1 the system became unstable. With lower values of the ratio, the system eventually trended towards the minimisation 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 normalised error starts quite low, rises up and, when it does not become unstable, goes back down. This is

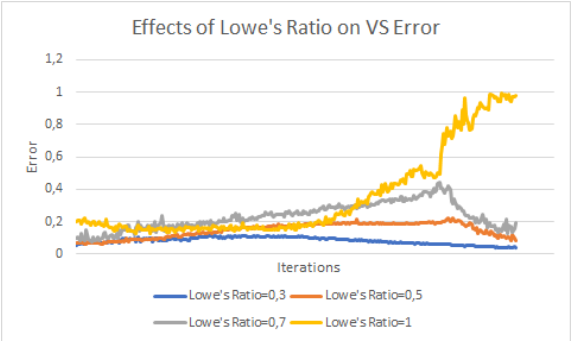
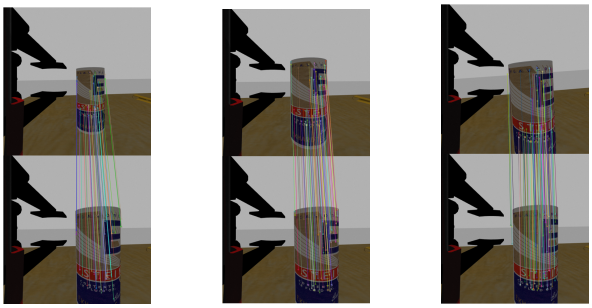


Fig. 9: Error of the visual servoing features with the variation of the ratio threshold.



(a) (b) (c)

Fig. 10: Frames at the beginning (a), middle (b) and end (c) of the visual servoing task. Top images are the current image and the bottom image is the target.

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 normalisation 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 number of features stabilises 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 10, one can see three frames taken during the servoing: (a) at the beginning, (b) at the midpoint and (c) near the target pose. The top images are the current view, the bottom ones are the desired one.

VI. Conclusions and Future Work

In this work, we have proposed a two-phase control scheme, where an open-loop controller drives the arm close to the target, and a markerless image-based visual servoing reduces the error on the final end-effector pose by closing the loop using images features. Our method does not assume any hand-crafted 3D model or any specific knowledge about the environment, apart from the assumption of a table on the scenario. The environment is created on the fly by means of 3D point cloud segmentation and clustering and a trajectory towards the target is calculated using Rapid Exploring random Trees. Moreover, and to reduce the final error, we developed a markerless image-based visual servoing strategy using SURF features which are calculated and evaluated (using Lowe's Ratio) in real-time.

As 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. The integration of visual servoing during trajectory execution would also be important. However, occlusions will always be a problem when it comes to cluttered environments since we rely on visual features. Furthermore, another important addition would be the incorporation of a depth estimation method. In this work, we have used a roughly estimated depth of the target features. Although it does not influence the task's success rate, it influences the speed of convergence of the visual servoing.

References

[1] S. M. LaValle and J. James J. Kuffner, "Randomized kinodynamic planning," *The International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, 2001.

[2] D. Kappler, F. Meier, J. Issac, J. Mainprice, C. Cifuentes, M. Wuthrich, V. Berenz, S. Schaal, N. Ratliff, and J. Bohg, "Real-time perception meets reactive motion generation," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1864–1871, 01 2018.

[3] T. La Anh and J.-B. Song, "Robotic grasping based on efficient tracking and visual servoing using local feature descriptors," *International Journal of Precision Engineering and Manufacturing*, vol. 13, no. 3, pp. 387–393, Mar 2012.

[4] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded Up Robust Features"," in *Computer Vision – ECCV 2006*, A. Leonardis, H. Bischof, and A. Pinz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 404–417.

[5] P. Vicente, L. Jamone, and A. Bernardino, "Towards markerless visual servoing of grasping tasks for humanoid robots," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 3811–3816.

[6] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, vol. 24, no. 6, pp. 381–395, Jun. 1981.

[7] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, Nov 2004.

[8] F. Chaumette and S. Hutchinson, "Visual servo control. I. Basic approaches," *IEEE Robotics & Automation Magazine*, vol. 13, no. 4, pp. 82–90, 2006.

[9] B. Siciliano and O. Khatib, *Springer Handbook of Robotics*. Berlin, Heidelberg: Springer-Verlag, 2007.

[10] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[11] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sendai, Japan, Sep 2004, pp. 2149–2154.

[12] I. A. Şucan and S. Chitta, "MoveIt!" [Online]. Available: http://moveit.ros.org

[13] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012.

[14] E. Marchand, F. Spindler, and F. Chaumette, "ViSP for visual servoing: a generic software platform with a wide class of robot control skills," *IEEE Robotics and Automation Magazine*, vol. 12, no. 4, pp. 40–52, December 2005.