Visual Servoing Project

Mahmoud Badran, Thien Bao Bui and Muhammad Arsalan Khawaja

December 28, 2020

Abstract

Visual servoing is one of the most important and growing field of Robotics and Computer Vision.This report provides a detailed explanation and implementation for the problem of Line following of robot and parking of robot using QR code as a parking sign. The report starts with the brief introduction and establishes the basic concepts of Visual Servoing used in this project to facilitate and prepare the reader for implementation part. The implementation has been carried out on 'The Construct' platform and local machine. The project has been experimented and supporting visuals and data are attached in the Evaluation part. In order to facilitate further detailed analysis, a GitHub repository has been prepared with proper documentation for executing and implementing the Visual Servoing tasks for Robot. Click **here** to navigate to GitHub repository.

Contents

Chapter 1

Introduction

A vision-based control system depends on continuous measurement of the target and the robot using vision. It creates a feedback signal and then moves the robot until the visually observed error between the robot and the target is zero [2]. The advantage of vision based control is the continuous measurement and feedback which provides great robustness with respect to any errors in the system. The Visual Servoing as described by Peter Corke in [2] is defined as:

> The task in Visual Servoing is to control the pose of the robot's end-effector, relative to the goal, using visual features extracted from an image of the goal object.

This section establishes the basics of Visual Servoing in order to be able to understand the visual servoing project. The mathematics for Position based Visual Servoing (PBVS) has been summarized. The different type of robots have been introduced before diving deep into the project. The chapter 2 describes the methodology and management of the project. A GitHub repository has been created in order to understand the project better and focuses on implementation and execution of code and environment. The chapter 3 discusses the implementation of the tasks for the project. It discusses the approaches and methodology for the execution of the project. The chapter 4 provides a detail analysis and evaluation of the project through rigorous and extensive experimentation. Finally the last chapter 5 provides a brief conclusion and recommendations that were observed and experienced for this particular project.

1.1 Types of Visual Servoing

There are two popular methods of Visual Servoing. They are described in figure 1.1 and described as follows:

1. **Eye in hand**: When the camera is mounted on end effector[1] observing the goal. Such configuration is called Eye in hand or end point closed

[1]The end effector is that part of the robot which interacts with the environment or target

loop. Eye in hand is further divided into two methods [5]. They are as follows:

(a) Image Based Visual Servoing (IBVS):There is no need to estimate the pose. The control is performed in image space by using image features directly.

(b) Position Based Visual Servoing (PBVS): The camera extracts the features of the goal object whose geometric information is already known. The camera has to calibrated precisely for this technique to work. This is computational expensive and is profoundly dependent on camera calibration accuracy and the model of the object's geometry.

2. **Eye to hand**: When the camera is fixed on fixed point in the World frame and is observing the goal as well as end effector. Such configuration is called Eye in hand or end point open loop.

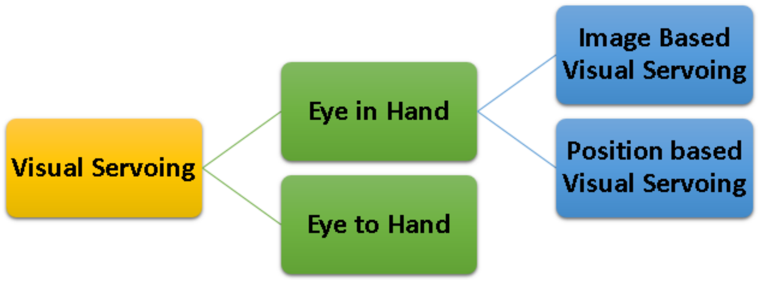


Figure 1.1: Flow Chart of Visual Servoing Techniques

In this project, PBVS was used. The closed loop control flow diagram is shown in figure 1.2.

or goal. It is referred to the device which is connected to end of the robot arm, where the hand would be.

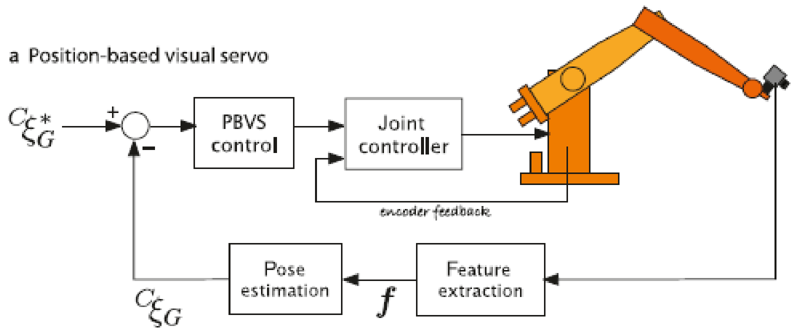


Figure 1.2: The closed loop control diagram of PBVS. Here ${}^{C}\xi_{G}$ is the estimated pose and ${}^{C}\xi^{*}_{G}$ is the desired pose of the goal. Image credits: [2]

1.2 Mathematical Background for PBVS

To estimate the pose of the goal object in PBVS, the camera calibration parameters are supposed to be known. The goal's geometric model is also supposed to be known.The relationships between the poses is demonstrated in figure 1.3.

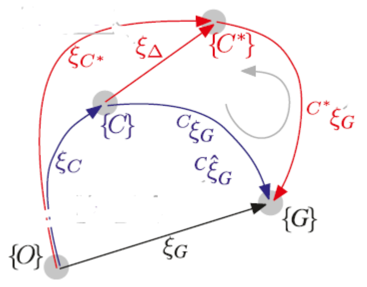


Figure 1.3: PBVS relative pose graph. Frame C is the current camera pose and frame C^* is the desired camera pose. The terms with 'hat' represent estimated values and terms with 'star' represent desired values. Image Credits : [2]

The desired relative pose is specified with respect to the goal ${}^{C}\xi^{*}_{G}$ and we wish to determine the motion ξ_{Δ} required to move the camera from its initial pose ξ_{C} to ξ^{*}_{C}. The actual pose of the goal ξ_{G} is not known. The indicated loop of the pose network is [2].

$$\xi_{\Delta} \oplus {}^{c^*}\xi_{G} = {}^{c}\hat{\xi}_{G} \tag{1.1}$$

where ${}^{c}\hat{\xi}_{G}$ is the estimated pose of the goal relative to the camera. It can be rearranged as:

$$\xi_\Delta = {}^{c}\xi_G \ominus {}^{c^*}\xi_G \tag{1.2}$$

This is the camera motion required to achieve the desired relative pose.The change in pose might be quite large so it is not reasonable to make this movement in one step. So, a point closer to $\{C^*\}$ is moved by:

$$\xi_C\langle k+1\rangle \leftarrow \xi_C\langle k\rangle \oplus \lambda\xi_\Delta\langle k\rangle \tag{1.3}$$

which is a fraction $\lambda \in (0,1)$ of the translation and rotation required.

1.3 Robot Operating System

Robotics Operating System (ROS), is a middle ware, low level framework, to write robotic software. It can be considered as an API to make the process of developing a robotic related projects more flexible, and simplified. There will be no need for an extensive knowledge of the hardware in which it saves much effort and time in the development phase. It includes many libraries and tools which connects and control the robot manipulators, handle the communication between multiple devices in a a working space. ROS is supported by many operating systems like Ubuntu, windows. Ubuntu is the more stable operating system working with ROS [3]. However, for the development of this project we are using the construct web platform, which is an online robotics working environment. The platform uses Ubuntu as the main operating system with ROS kinetic and uses the **Gazebo** as real world simulator with many robot model simulations.

The Project is divided into two main tasks. These are as follows:

1. Line following of the Robot
2. Parking of Robot.

1.4 Types of the Robot used

The different robots were used in this project in order to comprehend and test the robustness of the developed project. These are as follows:

1.4.1 ROSBot

ROSbot 2.0 is an autonomous, open source robot platform running on Husarion CORE2-ROS controller [4]. The ROSBot is shown in figure 1.4. The robot has following specifications:

- RGBD camera Orbbec Astra
- RPLIDAR A2 laser scanner.
- MPU 9250 inertial sensor (accelerometer + gyro)

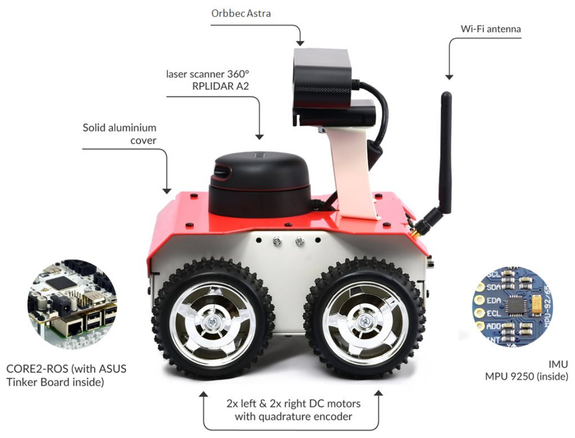

Orbbec Astra
Wi-Fi antenna
laser scanner 360° RPLIDAR A2
Solid aluminium cover
CORE2-ROS (with ASUS Tinker Board inside)
IMU MPU 9250 (inside)
2x left & 2x right DC motors with quadrature encoder

Figure 1.4: ROSbot 2.0

1.4.2 Turtlebot3 Waffle

TurtleBot3 Waffle by Robotis is without an doubt a very powerful robot for exploring ROS (Robot Operating System). It has ability to facilitate high payload and is equiped with additional sensors. It is shown in figure 1.5. The robot has following specifications:

- 360˚ Laser Rangefinder (LiDAR) for mapping, positioning (SLAM) and navigation
- Simple on-board computer (Raspberry Pi 3)
- OpenCR controller (32-bit ARM Cortex M7)
- Lithium polymer battery (Li-Po) 11.1V 1800 mAh
- Raspberry Pi Camera
- Bluetooth module for remote control

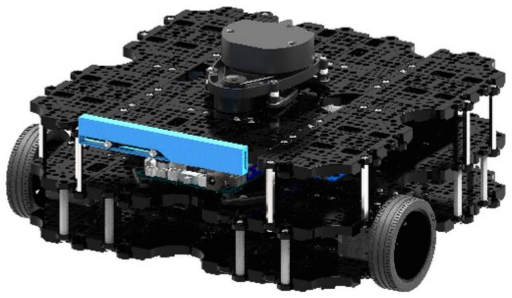


Figure 1.5: Turtlebot 3 Waffle by Robotis

1.4.3 Turtlebot 2

TurtleBot 2 is a low-cost, personal robot kit for educational purpose with open-source software. It has following specifications and is shown in figure 1.6. It has following specifications:

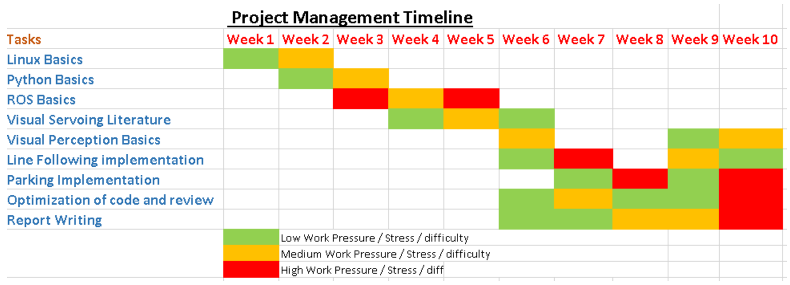
- Kobuki Base
- Kinect Mounting Hardware
- Asus Xion Pro Live

Figure 1.6: Turtlebot 2

Chapter 2

Project Management

The project was divided into two main tasks. Line following and parking. A significant amount of time was spent on learning the prerequisites and preparing the basic skills for ROS before starting the project. The figure 2.1 shows project timeline.



Project Management Timeline

Tasks	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10
Linux Basics	🟩	🟧								
Python Basics		🟩	🟧							
ROS Basics			🟥	🟧	🟥					
Visual Servoing Literature				🟩	🟧	🟩				
Visual Perception Basics						🟧			🟩	🟧
Line Following implementation						🟩	🟥		🟧	🟩
Parking Implementation							🟩	🟥	🟩	🟥
Optimization of code and review						🟩	🟧	🟩	🟩	🟥
Report Writing						🟩	🟩	🟧	🟧	🟥

🟩	Low Work Pressure / Stress / difficulty
🟧	Medium Work Pressure / Stress / difficulty
🟥	High Work Pressure / Stress / diff

Figure 2.1: The project timeline depicts the tasks and time spent on these tasks in order to complete the project. The constraint of time was gravely felt.

2.1 Constraints and Limitations

- Limited Time.
- System constraints (Lack of Powerful hardware).
- Lack of experience on ROS.
- No access to real physical robots due to COVID-19.
- The unstability of The Construct platform.
- No Classroom or lab experience due to confinment. The lack of guidance and lonliness was felt working in confined spaces.

- Tutorials on ViSP library and documentation was found out to be ambigiuos and at times missing the important information.
- Poor Internet access in Residence made our learning difficult.

2.2 Resources

- Ubuntu
- Kinect ROS
- Gazebo
- Construct
- OpenCV
- VISP Package
- Ar_track_alvar Package
- Blender for 3D modelling

2.3 GitHub Repository

Since the construct was having some problems with saving the progress, we use Github to manage and write technical review. Due to confinement, Its not appreciated to work toghether everytime, Github provides us great solution of working distantly through its fundamental facility of version control. Meetings among group members were only hold to discuss critical tasks.

The github repository can be found by clicking here. An extensive readme file guides through the tutorials for achieving the project objectives.

Chapter 3

Implementation

The implementation is the most critical part of the project. Python and C plus plus were used for ROS. Visual servoing has some powerful libraries and packages which help with the visual servoing tasks.

The project has been divided into two main tasks. **Line following** of the robot and **parking** of the robot. In order to implement it, It was quite challenging to write a code compatible with the environment. Enormous issue were felt while working on the online platform and simulation were quite unstable by the host website. Unfortunatley the Visual servoing could not be realized on the physical robots in the robotics lab due to confinement constraints imposed by the government due to COVID-19.

3.1 Task 1: Line Following of the Robot

The camera was installed on the top of the virtual robot in 'The Construct' platform. The camera takes images and sends it to a topic. The main task identified was to utilize these camera images for running some algorithm to achieve the line following of the robot. The popular package in ROS for doing this task is **OpenCV_bridge [1].** This package allows the conversion between ROS Image messages and OpenCV images. The following figure describes the role of the OpenCV_bridge package.

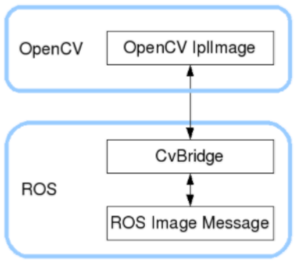


Figure 3.1: The package OpenCV_bridge converts images to OpenCV format which allows the user to process images for further puposes.

OpenCV images inherently come in BGR image format, whereas normal ROS images are encoded in standard RGB. Here, It is important to note that OpenCV_bridge provides a nice feature to convert between them and also provides many other privelges that can be utilized with OpenCV.

To get the images from ROS topic and showing them in OpenCV, a new package named as followline was created, with dependency on rospy. Two new folders were also created inside this package. They are named as follows:

- launch
- src

In the src folder, we create a python file named **followline.py**. The following command in this script converts the ROS image to OpenCV format.

```
cv_image = self.bridge_object.imgmsg_to_cv2
                              (data, desired_encoding="bgr8")
```

The image has been retrieved from a ROS topic and stored in OpenCV variable. The variable data in arguement is the one that contains a ROS message with the image captured by the camera. The map was imported from the Construct platform. We also tried to built up our own Map on Blender software but the construct software was unable to import it. The visuals of the robot initialized in the environment are shown in figure 3.2. The map from render is shown in Appendix.

The raw image is useless unless we preprocess it to take out all the useful information and remove/crop the non useful part of the image to make the program work faster. It's important to work with the minimum size of the image required for the task. Note that it is also critical to optimize the region of the image as a result of cropping. If it's too big, too much data will be processed, making program too slow. On the other hand, it has to have enough image to work with. At the end, one has to negotiate between computation and accuracy and find the best.

It is difficult to work in RGB environment, so one way to simplify this problem is to use HSV colour space because HSV removes the component of saturation thus reducing the complexity. The following command converts the colour space.

```
hsv = cv2.cvtColor(crop_img, cv2.COLOR_BGR2HSV)
```

In order to detect the line, the colour has to be selected in HSV space and defined. Since HSV values are hard to generate, it's better to use a color picker tool like ColorZilla to pick the RGB coding of the color to track. In this case, it's the yellow line in the simulated map. Finally, one has to select an upper and lower bound to define the region of the cone base that will be consider as Yellow.

In order to detect the colour, the mask is applied. The mask is nothing but the black and white version of the cropped image. Here the colour to be detected will be shown white and everything else black, thus simplifying the complexity of detection problem. The following command gives us the mask.

```
mask = cv2.inRange(hsv, lower_yellow, upper_yellow)
```

The arguements lower_yellow and upper_yellow are defined by user and threshold to detect the line.

To manipulate the robot keep following the line, the concept of centroids is used. Centroids, in essence, represent points in space where mass concentrates. This concept when immersed into image means instead of having mass, color is used. The place where there is more of the color that one is looking for is where the centroid is supposed to be. It's the center of mass of the blobs. To calculate centroid in binary black and white image is really easy.

With the information of centroid, the robot is controlled with the propotional control. The idea here is to keep the centroid in the centre of the image. Any error will be removed by applying propotional control to the robot model. The figure 3.2 demostrates the line following visuals to establish the successful completion of the task. The figure 3.3 shows the operations happening behind the successful tracking.

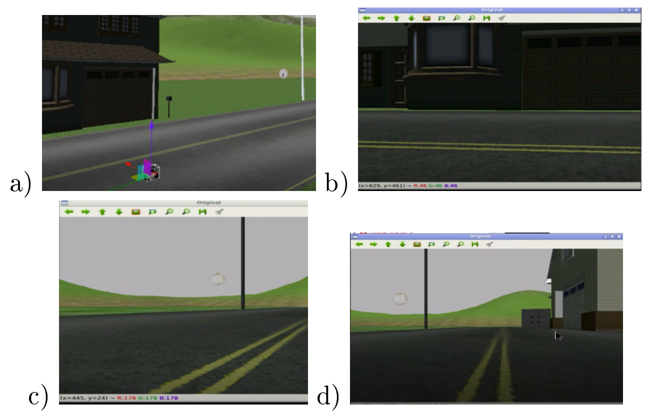


Figure 3.2: a) The robot has been initialized offside to the line. b) The camera visual image from the robot. c) After the robot started to follow line. It can be seen that it is moving towards line. d) The robot has been successfully aligned itself according to the line and it is now following line.

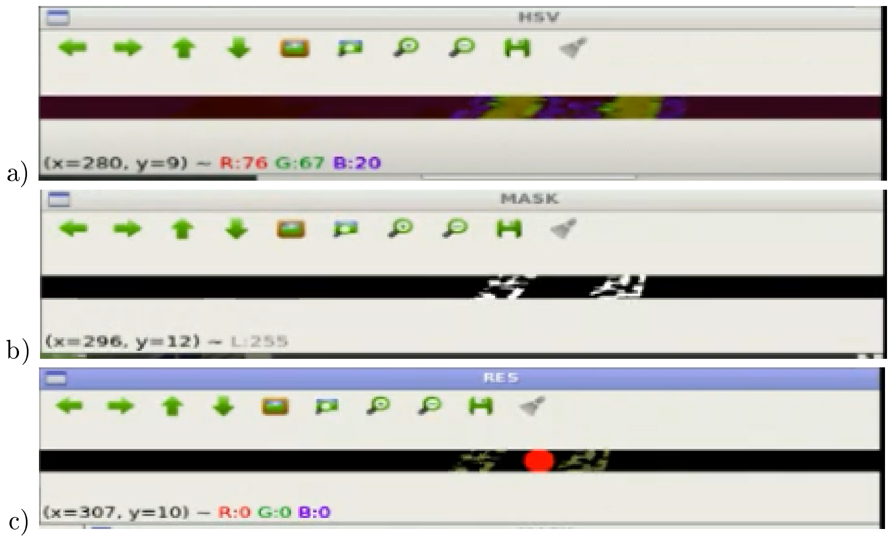


Figure 3.3: The preproccesing applied on the images as percieved by robot. The image has been cropped to save the processing time for line following. a) The cropped image in HSV image space. b) The mask of the preprocessing image. c) The RES image related to tracking algorithm. The red ball tries to be between the side lines.

The figure 3.4 shows the robot moving and following the robot.

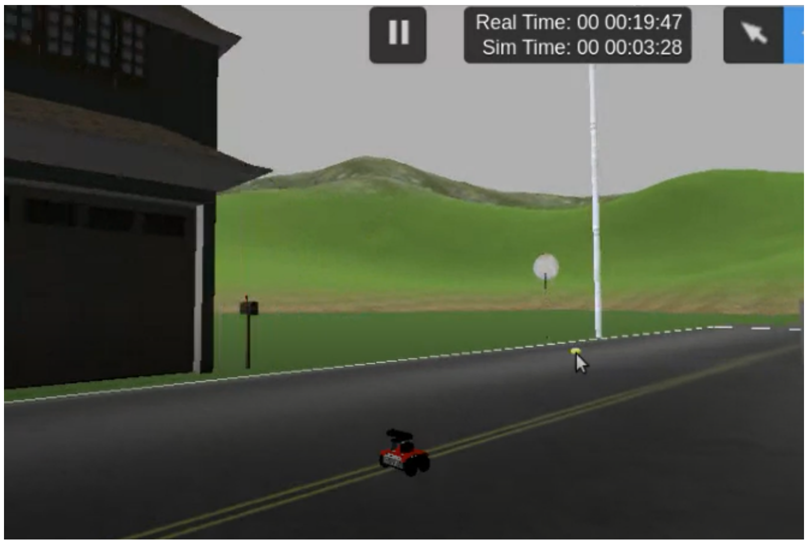


Figure 3.4: The robot can be seen following the line.

The task 1 of making the robot follow the line has been completed. The code and files are available on the github repository. The detail videos are also available on github to facilitate further analysis.

3.2 Task 2: Parking of the Robot

The Parking of the robot is a complicated task which involoves a lot of knowledge and programming skills. The parking task was successfuly completed with third approach after two failed approaches. All of approaches will be described here. The unstability and non-compatibility of The Construct platform was gravely felt. The figure 3.5 gives the bird eye view of our work on this task.

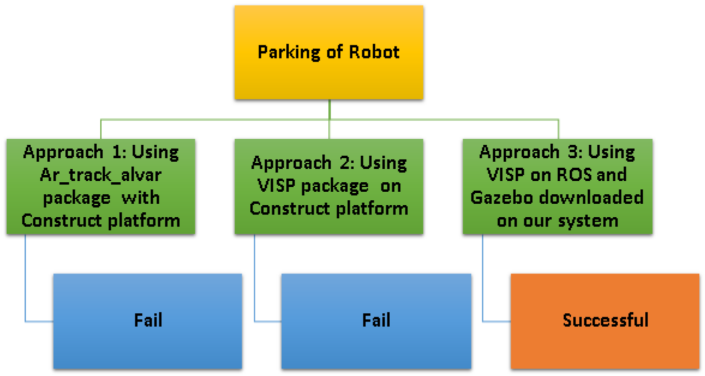

Parking of Robot
Approach 1: Using Ar_track_alvar package with Construct platform
Approach 2: Using VISP package on Construct platform
Approach 3: Using VISP on ROS and Gazebo downloaded on our system
Fail
Fail
Successful

Figure 3.5: The summarized approahes for achieving parking task.

The ROS and Gazebo was downloaded on our local system in desperation to make things work. It put a lot of difficulties for our laptop to run the heavy softwares. Overheating and system collapse were reguraly observed. The approches are explained in detail in the next sections.

In the initial steps, the images from the robot were fetched along with the depth information from the RGB-D camera of the camera. This important data is then processed by library function from either Ar_trac_alvar package or Vision ViSP package [3]. These libraries have functions that help us detect the AR tag or QR code which helps robot guide. The output of the algorithm or Visual servoing task is the velocity of the robot which decides wether robot should keep on moving or should stop.

3.2.1 Approach 1

The approach 1 uses Ar_trac_alvar package which is one of the most popular package for visual servoing applications. An AR tag was successfuly created but somehow it does not display correcly on map. In order for the robot to park, the robot needs to detect the AR tag but unfortunately the robot could not detect any AR tag whatsoever. We tried quite hard trying to trouble shoot the problem but we failed to make it work. It was concluded and realized that the construct platform has some fundamental limitations. It remains silent on maps, model and their code can not be found to make improvment to analyze map. It was also found that there server has also put some limitations when we were importing or using some extra files. However, this is our honest opinion and observation and we are unaware of the fact. The flowchart shown in figure 3.6 summarizes the approach.

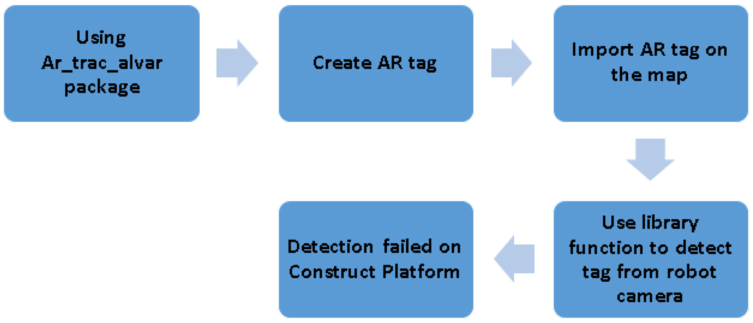

Using Ar_trac_alvar package
Create AR tag
Import AR tag on the map
Use library function to detect tag from robot camera
Detection failed on Construct Platform

Figure 3.6: Flow Chart of the first failed attempt

3.2.2 Approach 2

With the first approach failed, An attempt was made to use some other package which might be able to work on the Construct platform. ViSP package was found out to be an alternative. In order to complete the task, we created QR code and imported it on the map. The same problem was again faced and the attempt was a mere failure and gave null results. It was now confirmed that the environment has compatibilty issues and in despertion we moved to install ROS and Gazebo on our local machines. The flowchart shown in figure 3.7 summarizes the approach.

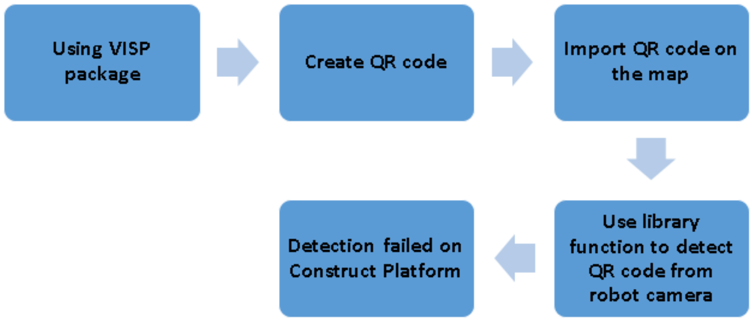

Using VISP package
Create QR code
Import QR code on the map
Use library function to detect QR code from robot camera
Detection failed on Construct Platform

Figure 3.7: Flow Chart of the second failed attempt

3.2.3 Approach 3

In this approach the environment was prepared on local machine. The map was uploaded. The QR code was generated and imported on map in ROS. Then the robot was initialized. The ViSP library was used. This package has quite some powerful function which make visual servoing possible and easy. The ViSP function recives the pose[1] information of the goal or target object

[1]Pose information refers to orietation and translation of a n object in the frame of reference

from the topic.This translationa and rotational information is transformed into homogenous matrix. Since the robot is using RGB-D camera, the robot also has privelge of providing the depth information of the goal or target which in our task is QR code. Finally the control law computes the velocity of the robot and publishes to to the dedicated topic. This velocity determines wheter robot shouls move or stop, accelerate or turn. The flow chart in the figure 3.8 summarizes the implementation. The code and supporting videos are attached in github repository.

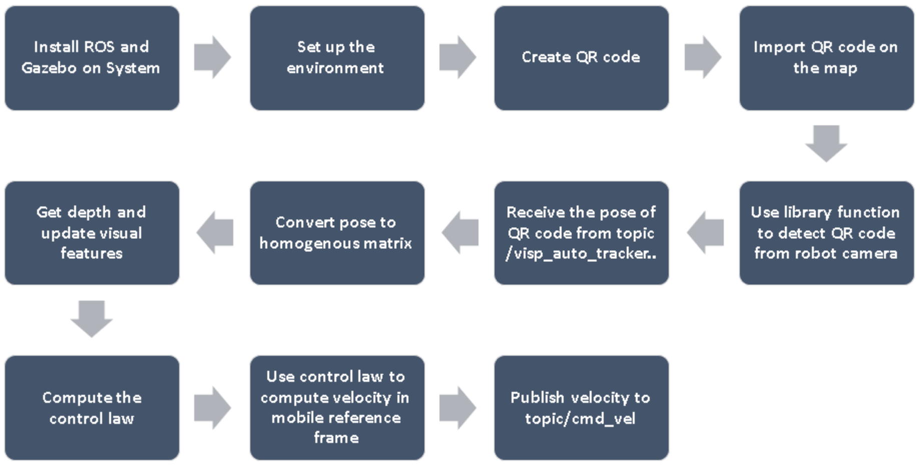

Install ROS and Gazebo on System
Set up the environment
Create QR code
Import QR code on the map
Use library function to detect QR code from robot camera
Receive the pose of QR code from topic /visp_auto_tracker..
Convert pose to homogenous matrix
Get depth and update visual features
Compute the control law
Use control law to compute velocity in mobile reference frame
Publish velocity to topic/cmd_vel

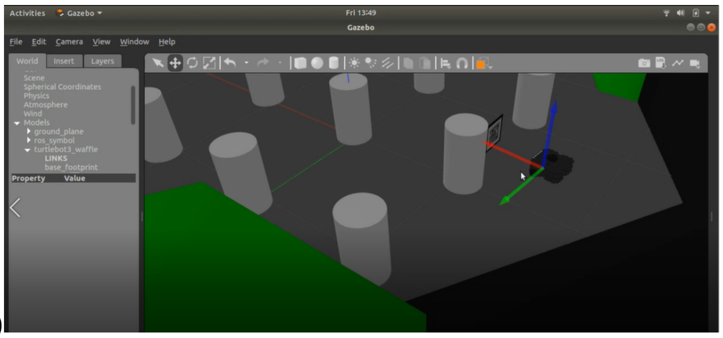
Figure 3.8: Flow Chart of the third successful attempt

Since we are using the Position based Visual Servoing(PBVS) method, the camera intrinsics should be known for PBVS to work as discussed in chapter 1. The Camera Internal Parameters are found and noted in table 3.1.

Camera Parameters	Description	Value in Pixels
P_x	Focal length in pixels (horizontal)	1206.889772
P_y	Focal length (vertical)	1206.889772
u_o	Optical centre (horizontal)	960.5
v_o	Optical centre (vertical)	540.5

Table 3.1: Intrinsic Parametes of Turtlebot 3 Waffle

The figure 3.9 shows the sucessful demonstration of the parking of robot.

a) 

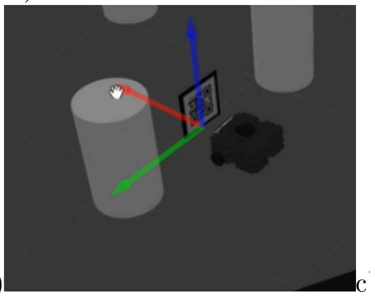
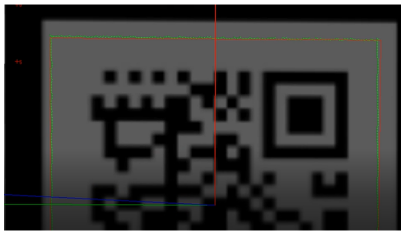
b)  c) 

Figure 3.9: a) The Turtlebot 3 Waffle has been imported into map along with QR code, which stands as a signal for parking. b) The robot can be seen approaching the QR code for parking and it stops after the required distance between the robot and QR code has been reached. c) The QR code as seen by robot camera and is being detected by camera. The complete video is available on github repository.

Chapter 4

Experimentation and Evaluation

4.1 Testing in different Maps

In order to validate and test the line following strategy, different maps were used to realize the robustness of the algorithm. The first map to test the algorithm is shown in figure 4.1.

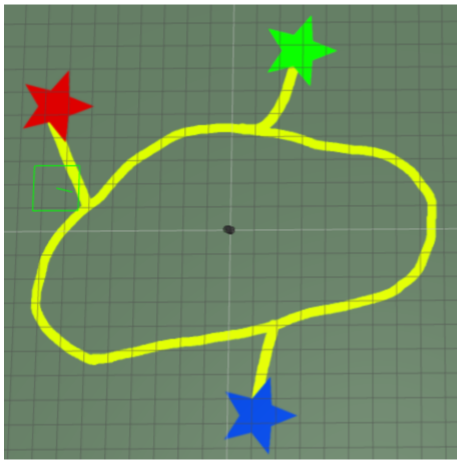


Figure 4.1: The yellow line is the line to be followed. The robot is initialized at the centre on the image, depicted by black dark dot.

The video of the robot following the line is available at github repository. Some of snaps to establish the successful following of line in this map is shown in figure.

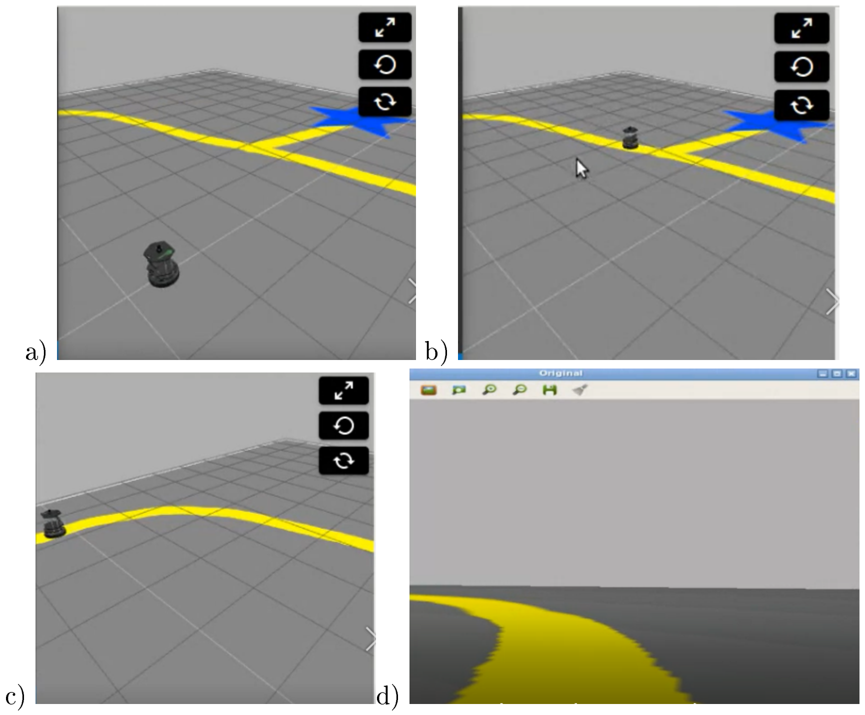


Figure 4.2: Snaps of robot following the line. a) The robot has been initialized. b) The robot has succussfully started to follow the line. c) The robot continues to follow the line. d) The images captured by camera mounted on robot.

4.2 Testing with different Robots

In order to check the robustness of the algorithm, Different robots were used for parking task. The result was successful. The figure 4.3 shows Task 2 with rosbot. The Turtlebot 3 waffle test has been mentioned and demonestrated in section 3.2.3. The detalied videos are available on github.

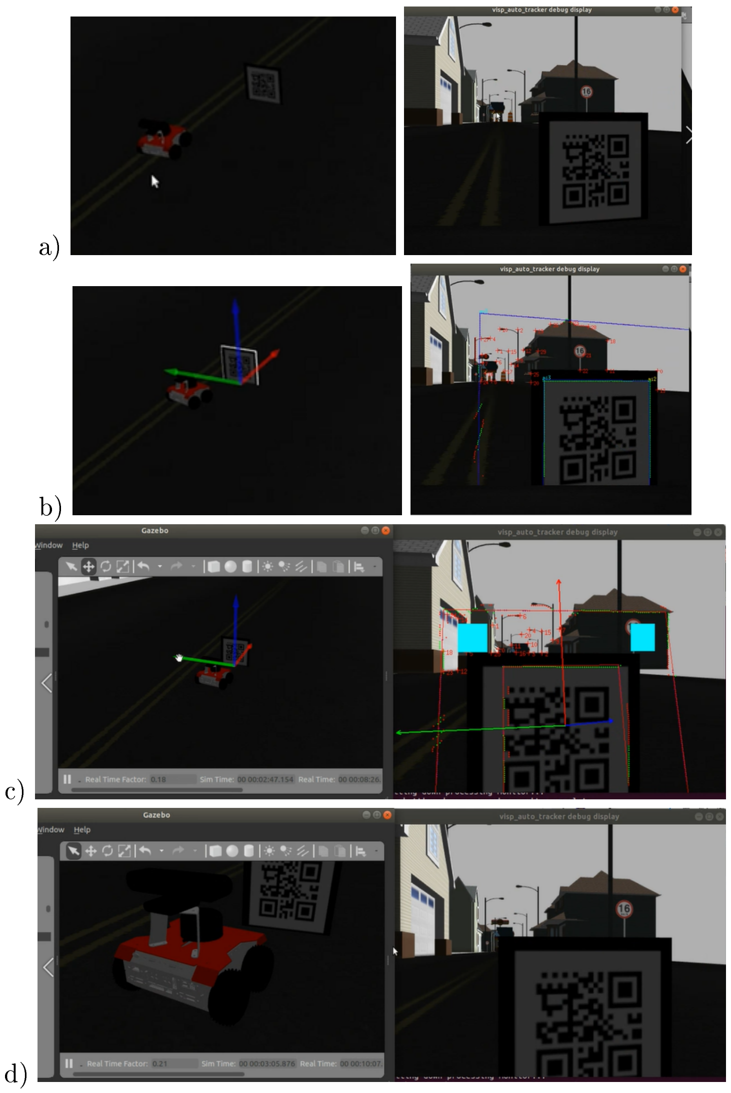
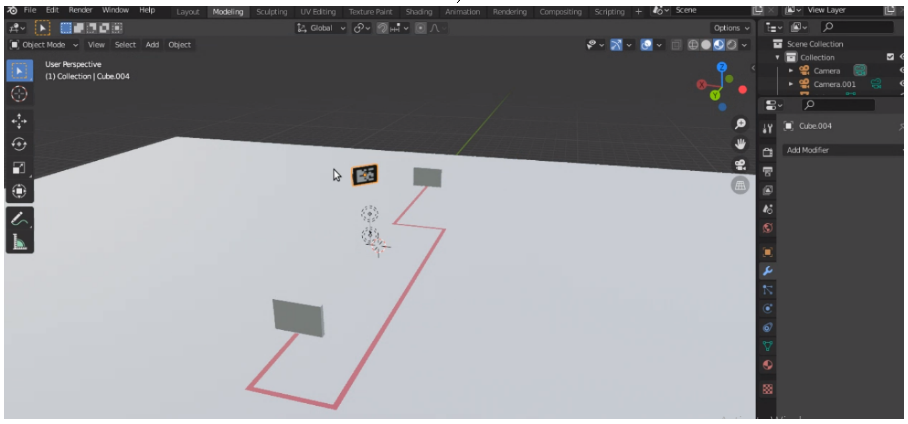

a)
visp_auto_tracker debug display
b)
visp_auto_tracker debug display
c)
Gazebo
Window Help
Real Time Factor:
Sim Time:
Real Time:
visp_auto_tracker debug display
d)
Gazebo
Window Help
Real Time Factor:
Sim Time:
Real Time:
visp_auto_tracker debug display

Figure 4.3: The succesful demonstration of RosBot doing parking task. a) The robot has been initialized, QR ode has been imported in the map. The right figure shows the image captured by robot. b) The robot can be successfully seen making its way to the QR code for parking. The QR code can be seen detected. c) The robot approaching parking and about to stop d) The robot has stopped and with threshold of distance achieved, algorithm has stopped working. The detail video can be found at github repository.

4.3 Creating and designing map

In order to explore further and push our limits, we decided to create our own map. For this matter, a software called blender was found out to be useful. Blender is a free and open-source 3D computer graphics software tool set used for creating 3D printed models, animated films, visual effects, motion graphics, interactive 3D applications, virtual reality, art and computer games. The map designed on Blender is shown in figure 4.4.

a)



b)

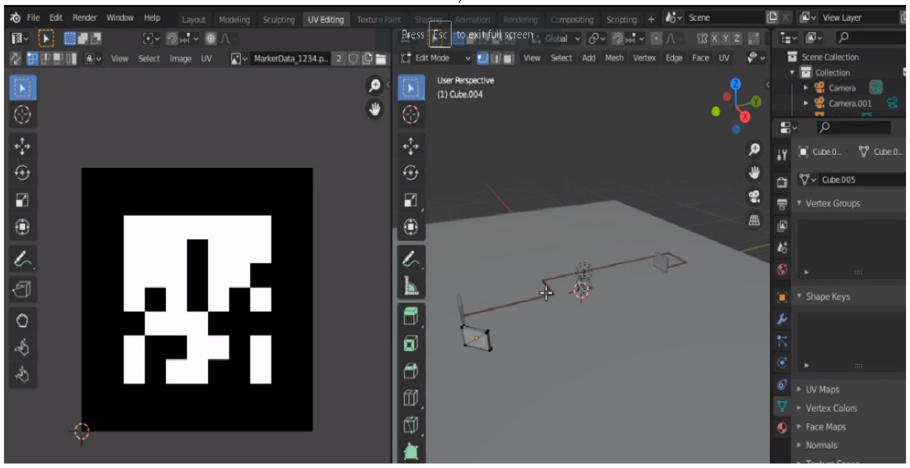


Figure 4.4: a) The figure shows the map created on Blender. The line supposed to be followed is the red line. The QR code has been also added. b) The figure depicts the adding procedure of QR code in order to facilitate task 2.

The QR tag was also added in the map to facilitate the second task. Unfortunately the map could not be imported to Construct because of limitation of the Construct platform and all of handwork in designing and learning this software went in vain. The Construct expressed a lot of compatibility issues.

4.4 Summary

In order to evaluate the performance of the project, A number of experiments were run for both the tasks, Line following and Parking of the Robot. Different Initialization, Illumination conditions in map, maps and robots were used to evaluate the experiments. The following pie charts in figure shows the success rate of both experiments.

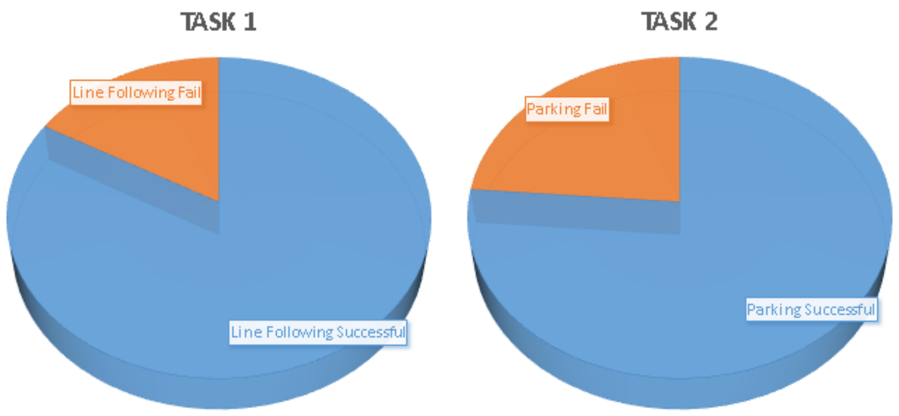


Figure 4.5: A series of 7 experiments were run for task 1. In most cases the robot was able to detect and follow the line. For task 2, 8 experiments were run, the failure rate was little high.

The reason for the failure in task 1 was mostly illumination conditions, change in color of the line. The reason for some maps being sometimes not compatible also fails the experiment. For task 2, it was observed that the main reason for the failure was not being able to detect QR code at far distance from the robot. The map imported on local machine also was incomplete. Sometimes the robot will fall into void and go missing suddenly and the experiment will fail

Chapter 5

Conclusion and Recommendations

This report provides a brief solution in the form of simulation for the Visual Servoing problem of Robot in an simulated environment. The simulation was run on Gazebo in 'The Construct' and also on the local system. The project is divided into two main tasks which are line following and parking of Robot. The three most important and useful packages used for this specific project are Visual VISP(Visual Servoing Platform), Ar_track_alvar and OpenCV_bridge. These packages are fundamental to Visual Servoing. Multiple Maps were tested provided by 'The Construct' in multiple courses and also an attempt was done to use our own designed map which was designed on Blender Software. The Construct courses were helpful and guiding in understanding the problem and implementing the solution. With the evidence of working simulations and visualizations, we are confident that if we apply our algorithm and code on the real robot, we will be successful in Visual Servoing the robot. We are also confident in integrating different ROS concepts and packages on Robots for Visual Servoing. However, we solemnly hereby do not claim that our solution is efficient, optimized or validated by practical experimentation or expert. We are open to any new ideas and we intend to learn, relish making mistakes and gain expertise in Robotics.

We are convinced that ROS is a very powerful tool in robotics however it is not easy to gain expertise. It is a universe of information in itself. We felt a lot of pressure for our tight schedules and the constraint of timing was felt gravely. We tried our best to practice and learn and what we have is a birds eye view of ROS and Visual Servoing. The Construct is a good platform and it makes learning of ROS quite smooth and efficient. However, it has some fundamental issues that make it less compatible. The Construct has stability issues. It is slow and sometimes it crashes. It is unstable, not trust-able regarding the privilege of saving progress. The Construct consumes a lot of Data and it is very difficult to use it on limited internet packages. We recommend them that there is a lot of room for improvement in computation and stability of website. We also recommend that the Visual Servoing project be done in the succeeding semester after ROS Robotic project. Since ROS is pre-requisite for this particular Visual Servoing project, we felt a lot of inconvenience, pressure, project management issues which adversly affected our performances dealing with both projects in the same semester.

Due to COVID-19, we are working from home and do not have any access to real robots and we understand that. But, It is certain that we might not be able to gain expertise in working with real robots and we will surely miss that opportunity.

Bibliography

[1] Kumar Bipin. *Robot Operating System Cookbook: Over 70 Recipes to Help You Master Advanced ROS Concepts*. Packt Publishing Limited, 2018.

[2] Peter Corke. *Robotics, vision and control: fundamental algorithms in MATLAB® second, completely revised*, volume 118. Springer, 2017.

[3] Patrick Goebel. *ROS By Example*. Lulu, 2013.

[4] Stanford Artificial Intelligence Laboratory et al. Robotic operating system.

[5] Ricardo Tellez, Alberto Ezquerro, and Miguel Angel Rodriguez. *ROS NAVIGATION IN 5 DAYS: Entirely Practical Robot Operating System Training*. Independently published, 2017.