

An overview of Visual Servoing Implementation

Supervisor

- Omar TAHRI

Presentation members

- Omar TAHRI
- Ralph SEULIN
- Marc Blanchon

Group

- Ahmed Hossameldin
- Vasileios Melissianos

Introduction

- Project's overview

01

Task 1

- Task overview
- Convert the images
- Apply image filters
- Move the robot

02

TABLE OF CONTENTS

Task 2

- Task approach summary.
- Map generation.
- Target initialization.
- Target recognition and localization.
- Parking approach.

03

Discussion

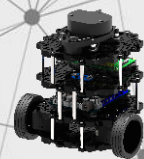
- Video demo.
- Conclusions.
- References.

04



01

Introduction



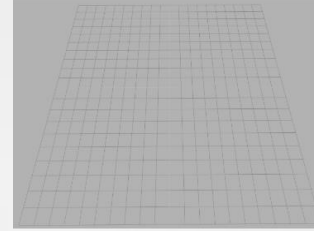
Project's overview

In this project, the aim is to develop an automatic system by monitoring the movement of a robot with an **eye-in-hand camera** in a specific course in Construct ([page](#) - Vision Basics Follow Line). The project's files and [video](#) are provided. The projects' challenges are provided, and the detailed summary is included in the next chapters.

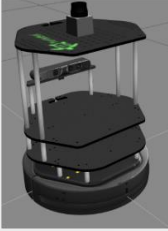
Challenges:

- Using the camera mounted on the robot, make the robot **follow** the map's line.
- Using the camera mounted on the robot, make the robot **find** the parking spot and **stop** at it. Use any specific pattern (QR code, bar code, parking sign, etc.) to mark the parking spot.

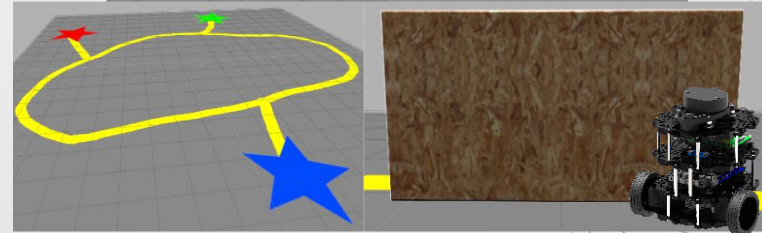
Map



Robot model



Objects





02

Task 1



Task overview

The aim of this task is to move the robot until it reaches the yellow line then follows it, using the camera mounted on the robot. In case the robot does not detect any line, it will turn around until it detects the line.

In order to approach this task, we divided it into three subprocesses as:

1. Get the images from the topic ("*/camera/rgb/image_raw*") and convert it to the OpenCV format using OpenCV_bridge package.
2. Apply some filters to the images to make operations faster and functional.
3. Move the robot based on the position of the centroid to follow the yellow line.

Used packages:

- The camera package [1]
- The cmd_vel package [2]
- The vision_opencv package [3]
- The cv_bridge package [4]





Convert the images

In this subprocess we will get the robot's image and convert it to OpenCV format then display it to get information from it.

At the first step, we have to know more about ("/camera/rgb/image_raw") topic and its message ("sensor_msgs/Image") as shown in the next figure.

```
user:~$ rosmmsg show sensor_msgs/Image
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

We can use ("echo") command to get each topic's output



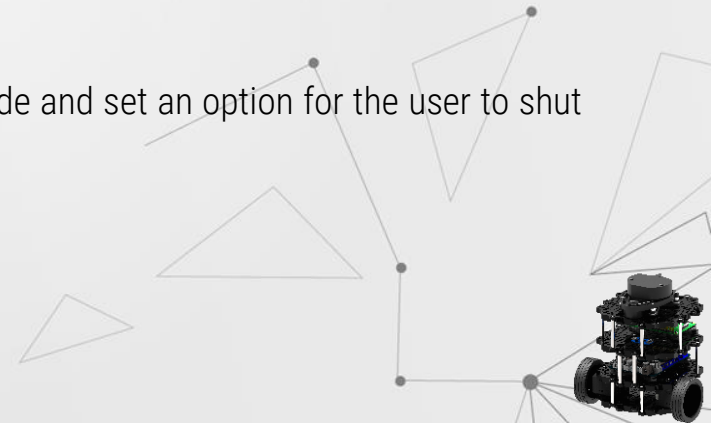


Convert the images

After we get the message information, we were able to write the python code that converts the images' format to be suitable for processing and display the camera output.

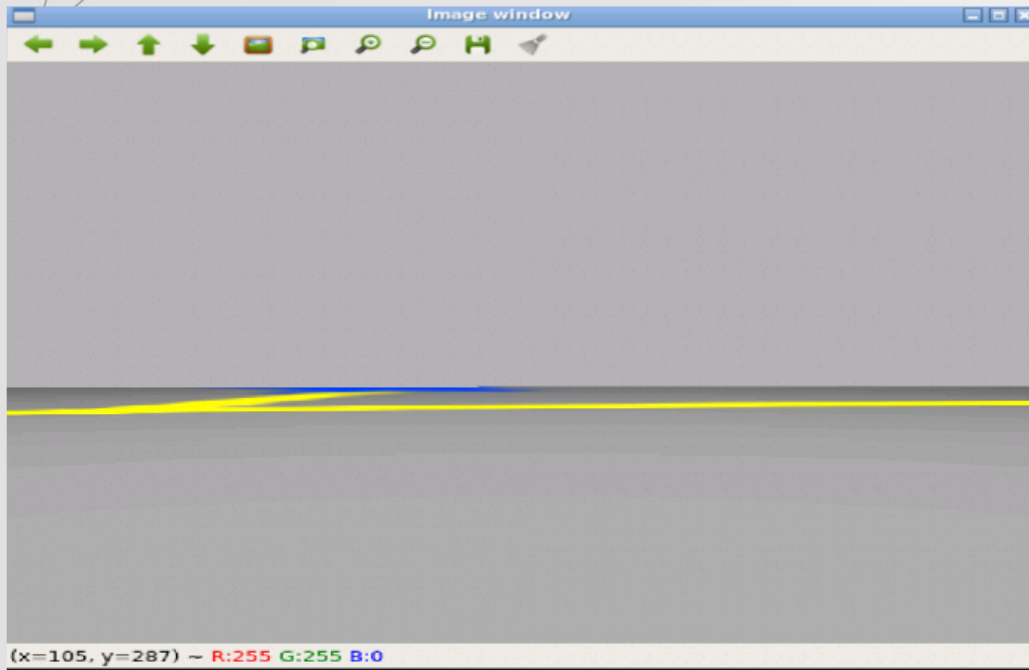
Firstly, we created a package and then created the python file (conv_img.py) that is divided into two parts:

- 1- The line follower class we created in it a subscriber node that gets the information from the topic, converting the image format into bgr8 because it is the OpenCV encoding by default, then displaying the image and getting all results by the callback function.
- 2- The main function that gets the live follower class results and put it in a node and set an option for the user to shut down the whole operation.



Convert the images

After we run the (conv_img.py) file, we will see the robot's camera's live image, as shown in the next figure.



We also see the position and the color of the mouse pointer point at the bottom bar of the image window.





Apply image filters

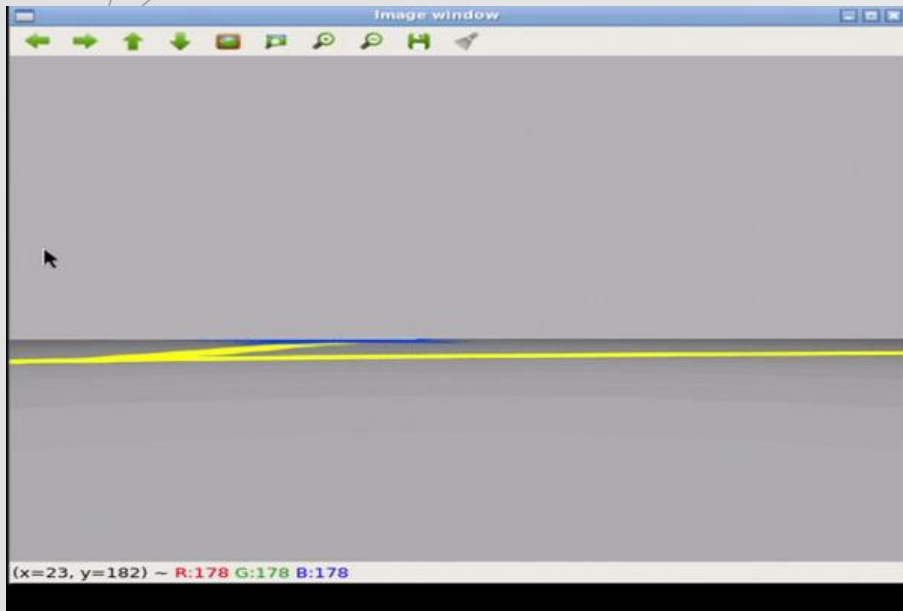
All we need from the image is the line that we want to track, so we applied four filters as follows:

- 1- The first filter crops the image part of the line to make the operation faster.
- 2- The second filter converts the colors into HSV format.
- 3- The third mask filter removes all colors except the line color only.
- 4- The fourth filter gets the centroid of the line and draws a circle on it.



Apply image filters

The first filter (cropping filter)



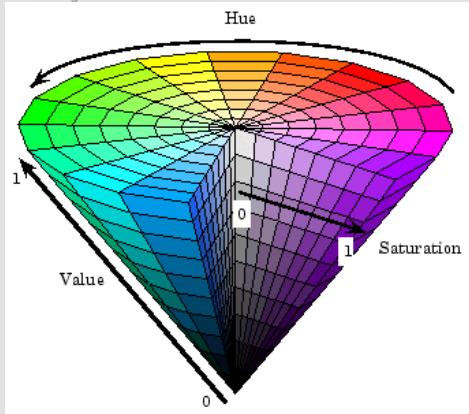
Using ("echo") command to ("/camera/rgb/image_raw/width") and ("/camera/rgb/image_raw/height") topics we got the outputs of these topics, the image dimensions are 460*640.

From the previous result, we can know what the target area is. We noticed that from pixel $y=0$ to $y=260$ is not useful, so we removed this top area. Then we cropped the area from the pixels $y=260$ to $y=440$, all the width pixels from $x=1$ to $x=640$ and removed the rest.

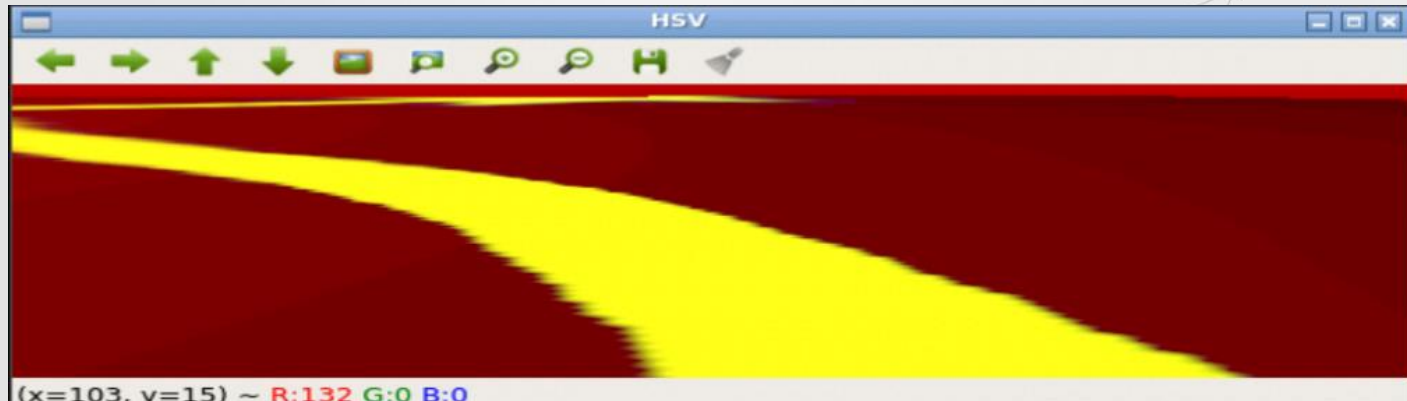


Apply image filters

The second filter (converting color format filter)



we have to make the robot detect the line's color in all lighting conditions. That will be by converting the BGR to HSV. HSV format considers all lighting conditions that face one color as a single color, but the BGR considers them different colors. This step is not that important in simulation because the lighting conditions are the same all time, but it will be essential if we have a real robot [8]. We wrote a simple code to convert the cropped image to HSV format using the cv2 package.



Apply image filters

The third filter (removing colors filter)

From the previous window result, the yellow RGB color is (255,255,0), so the BGR is [0, 255,255].

The HSV format of the yellow color is [[[30 255 255]]]. We must put limits (max and min) of the yellow region (to make the robot consider all this region a yellow color). The region will be between (yel_min = [[[20 200 200]]] and yel_max = [[[40 255 255]]]). After that, we will apply a mask in the HSV cropped image to make it black and white. The white will be the colors between the yellow limits, and the black will be the rest colors. That will make the detection and processing too easy because it is only two options (black or white). We used the command ("cv2.inRange") to get only yellow colors and the command ("cv2.bitwise_and") to merge the mask with the cropped image.



Apply image filters

The fourth filter (getting the centroid of the line and draws a circle)

We used the command ("cv2.moments") that calculate the moments of the binary image and then we got the centroid coordinates of the target line yellow color using the moments results.

The moments of any image ($I(x, y)$) is:

$$M_{ij} = \sum_x \sum_y x^i y^j I(x, y) \quad (1)$$

The central moments will be as:

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q I(x, y) \quad (2)$$

So, the centroid of the image will be:

$$\bar{x} = \frac{M_{10}}{M_{00}}, \bar{y} = \frac{M_{01}}{M_{00}} \quad (3)$$

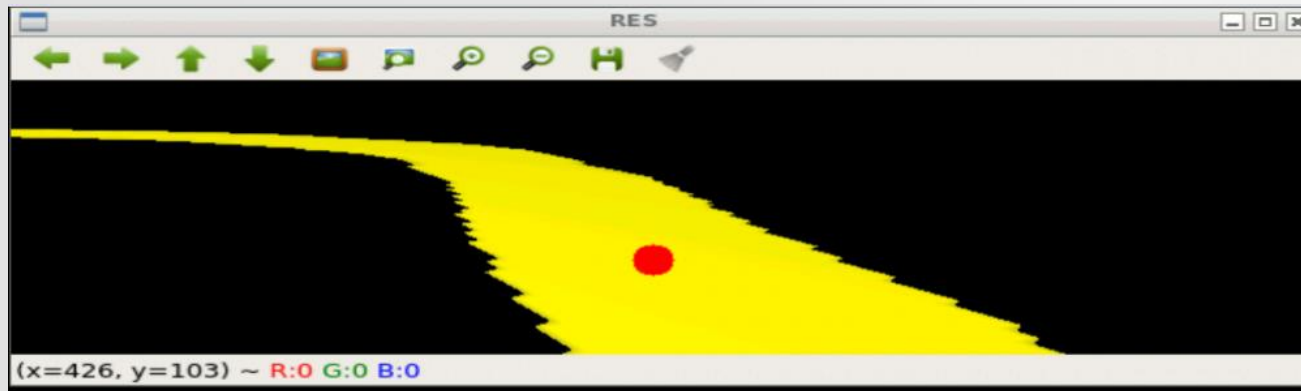
We used equation (3) to get the centroid of the yellow line [9].



Apply image filters

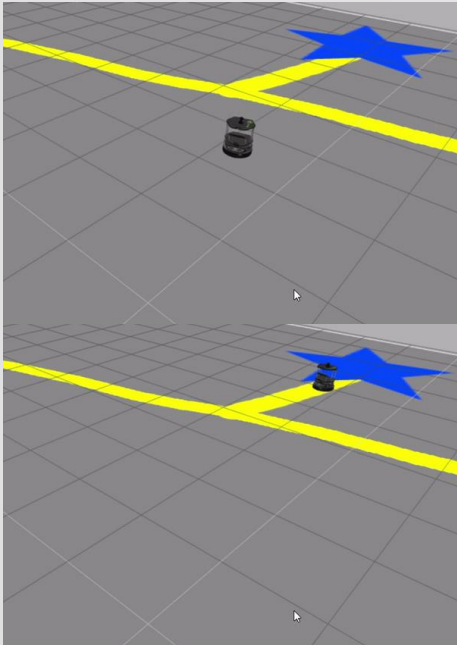
The fourth filter (getting the centroid of the line and draws a circle)

Then we drew a red circle on this circle by using the command ("cv2.circle").



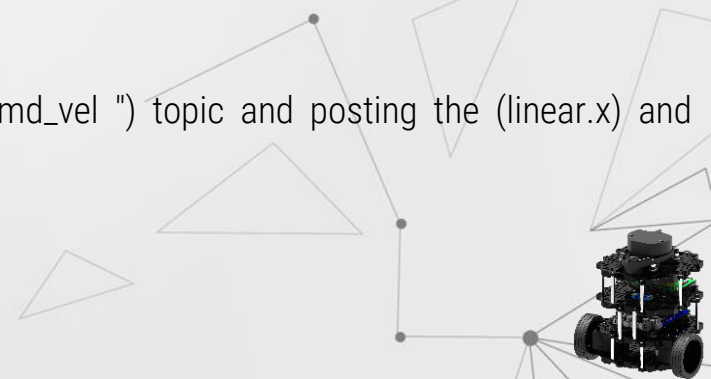
Move the robot

The last step is to move the robot according to all the previous actions. The control of the movement is passed on a Proportional control. That maybe had a small error, but it is one of the best ways to make a line follower.



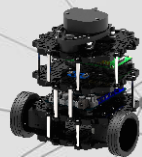
The idea of the code is to always gives a constant linear motion if the camera detected any yellow line and gives the angular Z velocity that depends on the distance between the centroid point of the detected line (the drawn red circle) and the center of the image then prints the info of the angular value. And turn the robot around (give only an angular velocity) if the camera didn't detect any yellow line (if there is Zero Division Error in the moments detecting).

The moving of the robot was by using (" /cmd_vel ") topic and posting the (linear.x) and (angular.z) values.



03

Task 2



Task approach summary

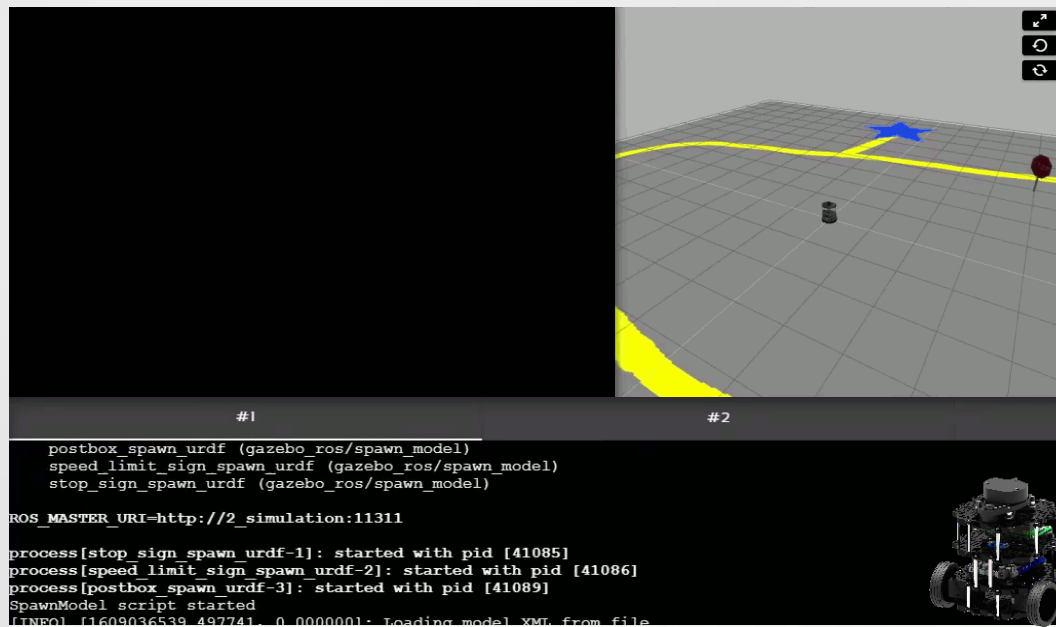
- Using the camera mounted on the robot, make the robot find the parking spot and stop at it. Use any specific pattern (QR code, bar code, parking sign, etc.) to mark the parking spot.

In order to approach the Task#2 we got inspired from [here](#) and divided the task into sub-processes:

1. Map generation.
2. Target initialization.
3. Target recognition and localization.
4. Parking approach.

Used packages:

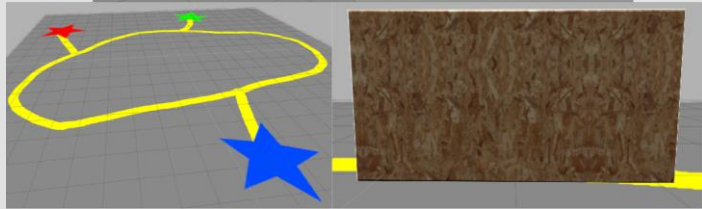
- The camera package [1]
- The cmd_vel package [2]
- The vision_opencv package [3]
- The cv_bridge package [4]





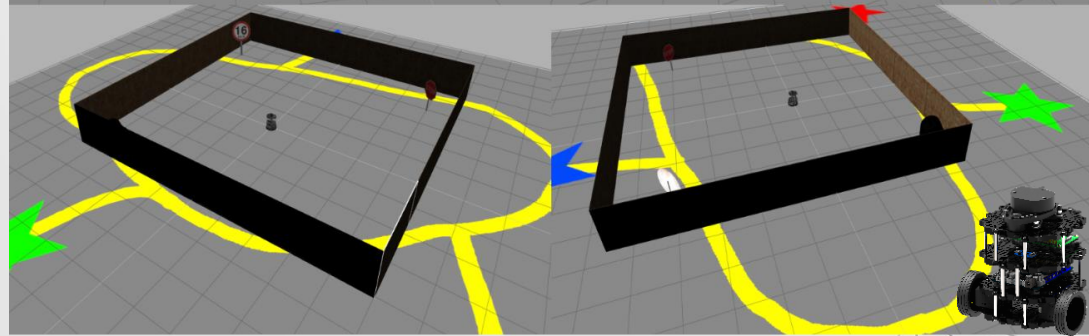
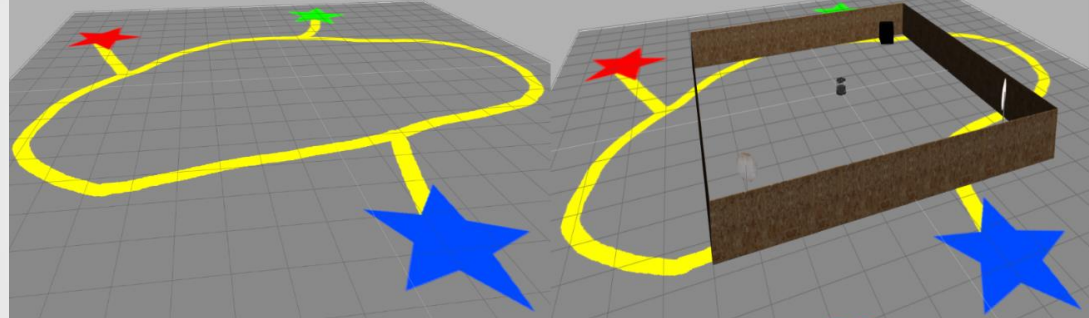
For the map generation we need a master-object-spawn file to generate the required objects for the map.

Objects



Map generation

```
user:~$ roslaunch image_pose_estimation spawn_object_master.launch
```

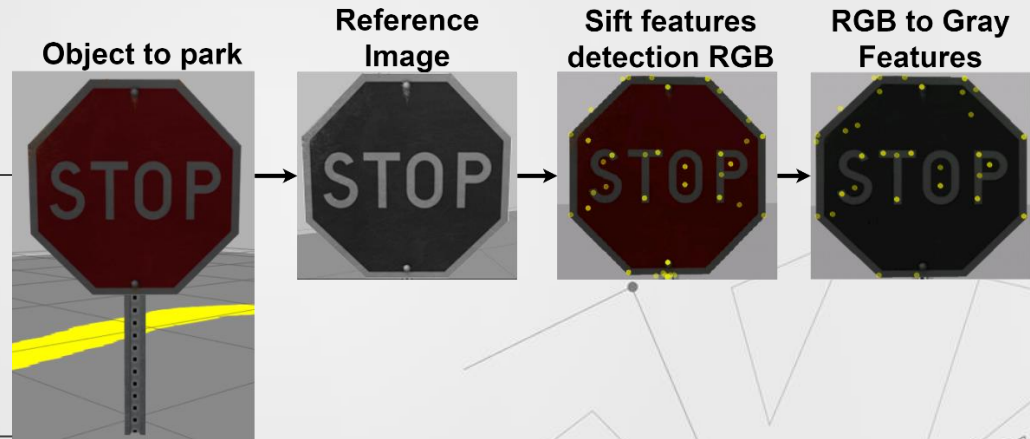


Target initialization

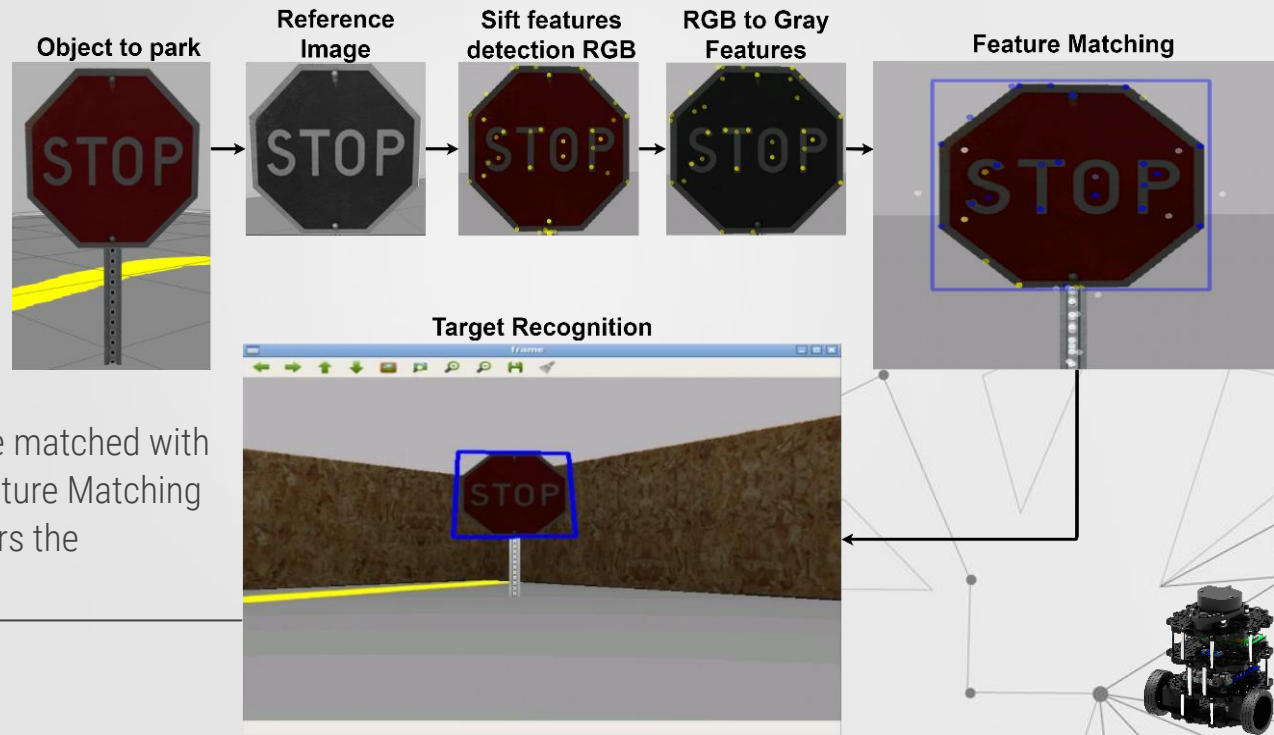
After generating the map, the next procedure is to initialize the reference image where the camera of the robot will recognize the object and give translation commands to park at it.

The generation of the reference image provides the input for the python file to be launched by the launch file which passes the robot's camera parameters, the blur threshold and the reference image path to the python's variable environment.

Then, features from the reference image are generated using the SIFT [5][6] algorithm for the purpose of having reference features for later matching.



Target recognition and localization



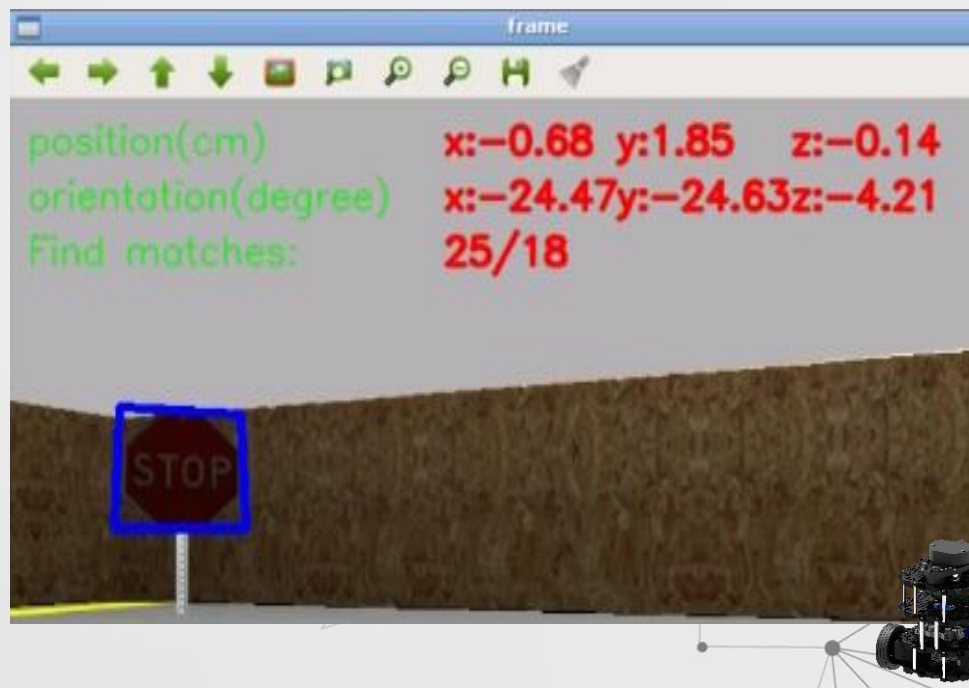
When the robot extract features that are matched with the reference features (blue dots in the Feature Matching image in Figure below), then the robot enters the recognition mode.

Target recognition and localization

The recognition mode refers to the pose estimation of the object with respect of the rectbox's corners and Homography matrix.

Homography is a planar relationship that transforms points from one plane to another. It is a 3 by 3 matrix transforming 3 dimensional vectors that represent the 2D points on the plane. In order to acquire these object's estimation points, we computed the Homography matrix using perspective transformation [7].

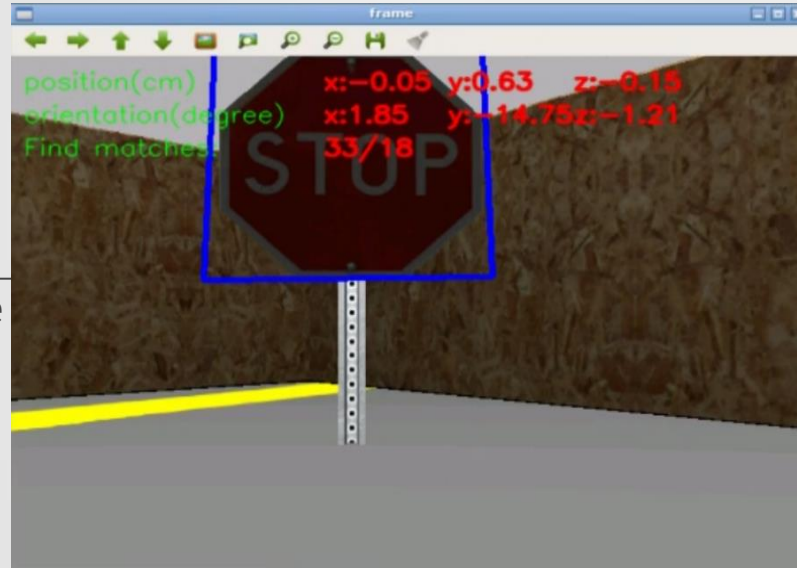
$$\begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} * \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} = H * \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix}$$



Parking approach

The acquired position of the object gives us the opportunity to move the robot as close as possible to this target. To achieve that, a simple yet efficient algorithm was established where the robot moves the object to the center of the frame and just moves forward.

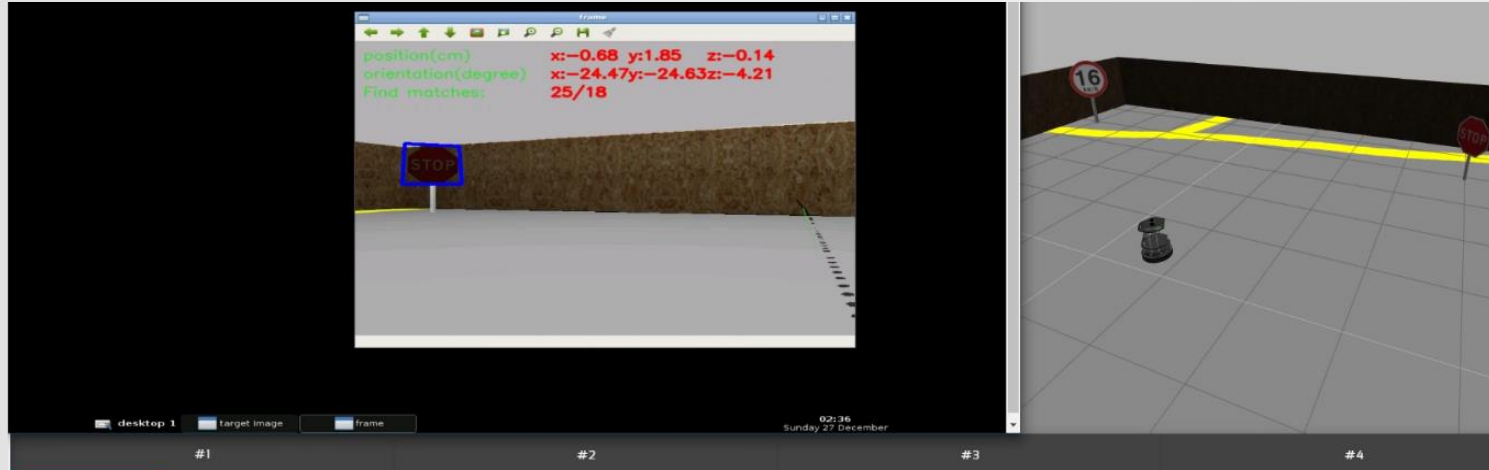
There are several case statements that move and center the robot to the object and one last statement that terminates the robot's movement if a specific position value is reached. "The main problem with this approach, is that the frame rate was too low, and we had to establish trial and error procedures to calibrate the movement of the robot."




```
Target Detected
Centering right
Centered!
Moving forward..
Target Detected
Centering right
Centered!
Moving forward..
Target Detected
Centering right
Centered!
Moving forward..
Target Detected
Centering right
The robot has parked!
Exiting..
```





Parking approach





```
Turning for target
Turning for target
Turning for target
Turning for target
Target Detected
Centering right
Centered!
Moving forward..
Target Detected
Centering right
Centered!
Moving forward..
Target Detected
Centering right
Centered!
Moving forward..
```

 Rotating on z axis for target detection.

 If the matches are greater than 18, then the target image is recognised.

 Rotating on z axis with small rotation value to center the robot with the target with respect to the object's X linear position.

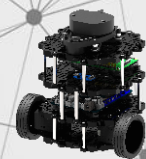
 When centered, it gives the attribute to move the robot forward.

 The robot moves forward until Y.linear object's position is in range of {0.33,...,0.55}.



04

Discussion



Video demo

The screenshot displays a ROS simulation environment. The main window is a 3D view of a robot (a black cylindrical base with a transparent top) on a grey floor. A yellow line is visible in the background. The interface includes a file explorer on the left showing the directory structure of a catkin workspace, with files like `CMakeLists.txt`, `package.xml`, and several packages. A terminal window at the bottom shows the command `roslaunch image_pose_estimation follow.py` being executed. The system tray at the bottom indicates the time is 4:00 PM on Sunday, 27 December.

Learn Robotics from Zero - Robo... X

app.theconstructsim.com/#/GraphicalTools

Learn Robotics from Zero - Robo... X New Tab

app.theconstructsim.com/#/Desktop

EXPLORER: USER

- catkin_ws
 - build
 - devel
 - src
 - feature_matching
 - image_pose_estimation
 - launch
 - models
 - src

CMakeLists.txt

package.xml

- my_blob_tracking_pkg
- my_object_recognition_pkg
- vision_visp

Ln 103, Col 41 LF UTF-8 Spaces: 4 Python

#1 #2 #3 #4

user:~/catkin_ws/src/image_pose_estimation/src\$ roslaunch image_pose_estimation follow.py

desktop 1

14:00
Sunday 27 December

3 - Vision Basics Follow Line

ROS Perception in 5 Days

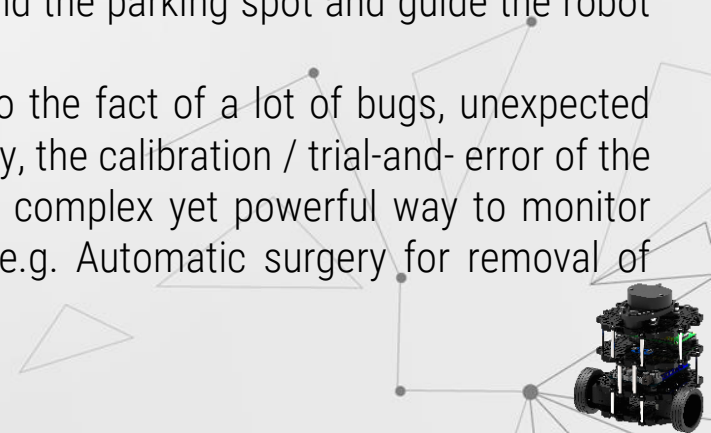
4:00 PM



Conclusions

We used a TurtleBot robot with an RGB camera in The Construct to simulating two tasks. The first task was to make the robot follow a yellow line. That was done by converting the image to use the OpenCV package then applying some filters to make the robot see the yellow line only and locate the center of that line and finally make the robot move according to the line center's location by using the Proportional control. The second task was to use the eye-in-hand camera to find the parking spot and guide the robot to park on it.

In general, the implementation of these tasks was challenging due to the fact of a lot of bugs, unexpected disconnections from the server, slow robot performance and finally, the calibration / trial-and- error of the robot with respect to its slow performance. Visual Servoing is a complex yet powerful way to monitor robots and its preciseness is very promising in difficult tasks (e.g. Automatic surgery for removal of malignant prostate).



References

- [1] "Cameras." [Online]. Available: <http://wiki.ros.org/Sensors/Cameras>.
- [2] "cmd_vel_mux." [Online]. Available: http://wiki.ros.org/cmd_vel_mux.
- [3] "vision_opencv." [Online]. Available: http://wiki.ros.org/vision_opencv.
- [4] "cv_bridge." [Online]. Available: http://wiki.ros.org/cv_bridge.
- [5] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," Int. J. Comput. Vis., vol. 60, no. 2, pp. 91–110, Nov. 2004.
- [6] V. D. Melissianos, "Classification of medical images with modern techniques of visual and image processing methods.," Technological Educational Institute of Crete, 2019.
- [7] Abhinav Peri, "Using Homography for Pose Estimation in OpenCV." [Online]. Available: <https://medium.com/analytics-vidhya/using-homography-for-pose-estimation-in-opencv-a7215f260fdd>.
- [8] Poorani, M., T. Prathiba, and G. Ravindran. "Integrated feature extraction for image retrieval." International Journal of Computer Science and Mobile Computing 2.2 (2013): 28-35.
- [9] Mukundan, Ramakrishnan, and K. R. Ramakrishnan. Moment functions in image analysis: theory and applications. World Scientific, 1998.



Thank you for your attention

