



## Visual Servoing

Masters 2

Team:

Vasileios Melissianos vm560566

Ahmed Hossameldin ah940806

---

## Introduction

Robot simulation through a compact and realistic environment, provides opportunity for robotics, CV (Computer Vision) and ML (Machine Learning) programmers a way of learning and enhancing their skills on a variety set of robot models and maps. Gazebo and ROS combined, provide these services for accomplishing numerous tasks in the Construct platform. Gazebo is responsible for the map, the objects, and the architecture of robots while ROS (Robot Operating System) having the role of controlling any kind of robot (mobile robots, robotic arms etc..) throughout a set of organized files also known as packages. 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.

## Project challenges

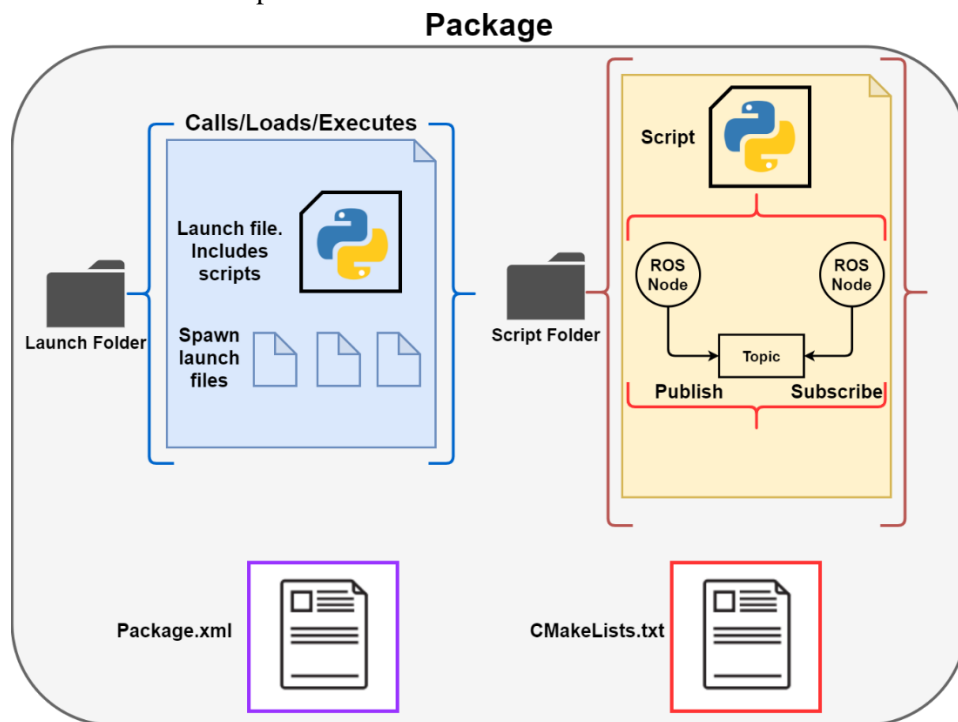
- Using the camera mounted on the robot, make the robot follow the map's line.
- Using the camera mounted on the robot, make the robot find the parking spot and stop at it. Use any specific pattern (QR code, bar code, parking sign, etc.) to mark the parking spot.

## ROS materials

To accomplish these tasks, the use of specific materials is necessary. The combination of these materials composes the appropriate package for a specific task. Thus, these components are summarized below:

- Packages: A package is a set of information that act like an independent library for a specific task. Precisely, a package provides a set of scripts, launch files, maps, configuration files, in order to get/give information from/to ROS nodes and thus, provide a full functional system for a specific task. A package provides:
  - ❖ Launch folder
    - Launch files: A launch file is a script that replaces the console commands and compose an automatic system that monitors all the file inside a package.
  - ❖ Script folder
    - Scripts: A script is an executable file used for accomplishing a task.
  - ❖ Package.xml: This unique file, contains a set of different packages to create a more customizable package (if necessary).
  - ❖ CMakeLists.txt: This unique file, contains some C++ fixed processes and building commands. This file is necessary for building a package.
- Nodes: A node is a built-in ROS function (python, C++) that provides/gets information from ROS topics. ROS nodes can subscribe or publish to a topic for the purpose of sharing important information such as metric values etc.
- Topics: A topic is the most important component in the package. It can transmit data between nodes (e.g. one node provides values to the topic, and another node prints the topic's values) and can provide data throughout ROS built-in sensors, functions etc.

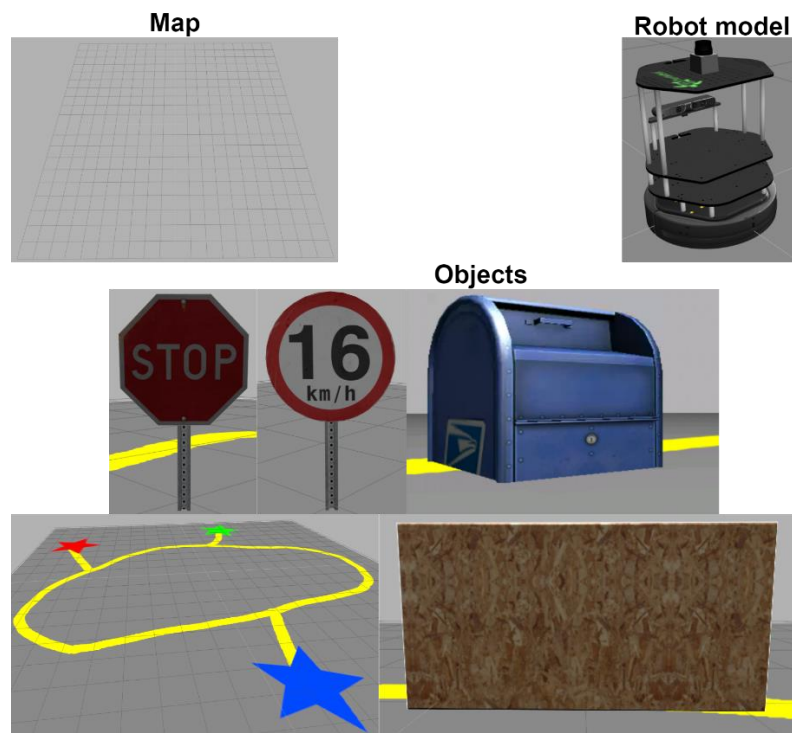
- Messages: A message is a data structure provided from a ROS node using a specific topic. These values either trigger the specific node or trigger some other nodes that are connected to this topic.



*Figure 1. A general concept of a ROS package*

### Gazebo materials

Gazebo also is used for spawning every task's [objects](#), the robot with the appropriate modules and the map where the robot will move on.



*Figure 2. A general concept of Gazebo services.*

## Task#1

- Using the camera mounted on the robot, make the robot follow the map's line.

Packages used:

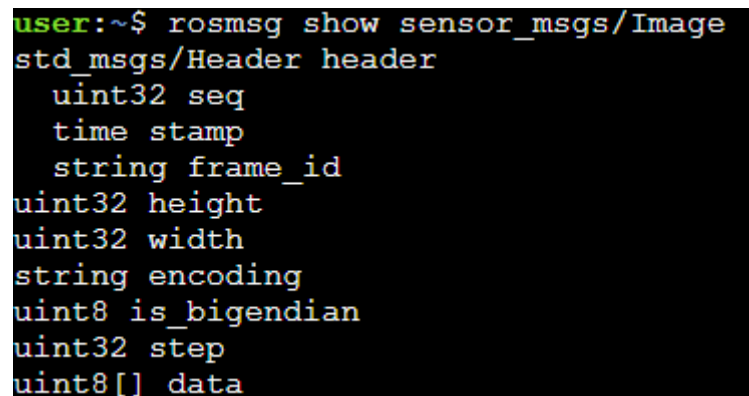
- The camera package [1]
- The cmd\_vel package [2]
- The vision\_opencv package [3]
- The cv\_bridge [4]

In order to approach the Task#1 we got inspired from [here](#) and divided the task into sub-processes:

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.

### Convert the images

At the first step, we had to know more about ("/camera/rgb/image\_raw") topic and its message ("sensor\_msgs/Image"). As we see in the next figure, the topic gets from the robot ten different messages.



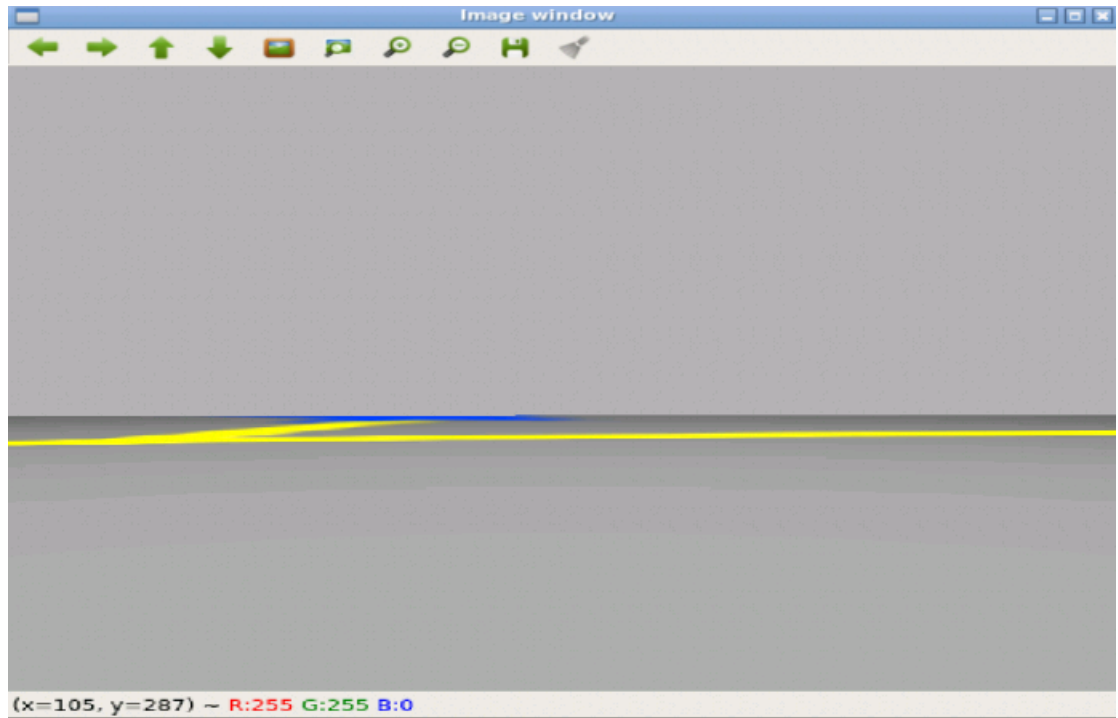
```
user:~$ rosmmsg show sensor_msgs/Image
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

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

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:

- 1- we applied the first filter to crop the image part of the line to make the operation faster.
- 2- Then applied the second filter that converts the colors into HSV format.
- 3- After that, we applied the third mask filter to remove all colors except the line color only.
- 4- Finally, we got The Centroids and draw a circle of the centroid and display all the live images.

Before applying the first filter, we had to know the dimensions of the images by getting the output of the topics ("/camera/rgb/image\_raw/width") and ("/camera/rgb/image\_raw/height"), 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.

To make the robot see only the target color, we had to get the line's color in all lighting conditions. That was 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.

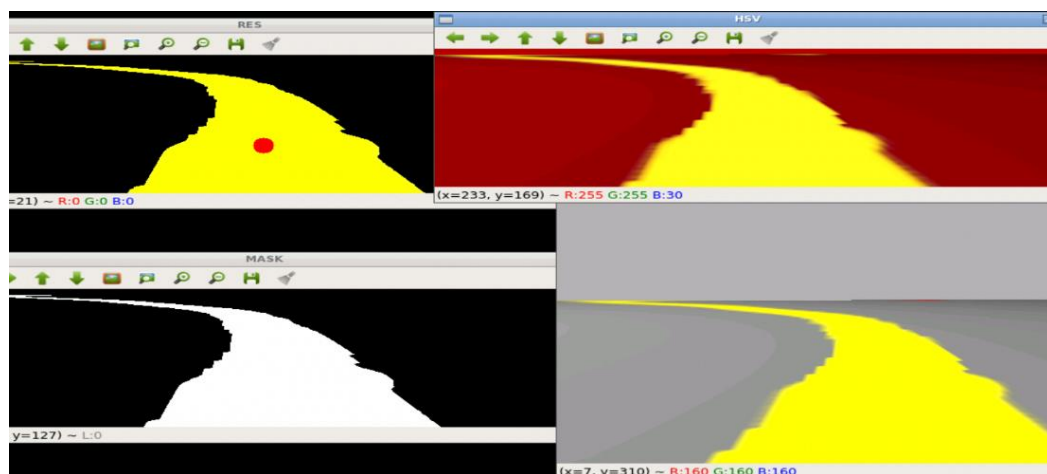
From the previous window result, the yellow RGB color is (255,255,0), so the BGR is [0, 255,255].

We wrote a simple code to convert the color to HSV format using the cv2 package. 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  $yel\_max = [[[ 40 255 255]]]$  ). And convert the cropped image into HSV format.

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.

Before moving the robot, the last step is to get the centroid of the detected line and draw a circle on that center.

We used the command ("cv2.moments") that calculate the moments of the binary image to get the centroid coordinates of the target line yellow color. Then we drew a red circle on this circle by using the command ("cv2.circle") and show all the filters outputs as shown in the next figure.



The figure shows:

- 1- The Black white mask output image.
- 2- The HSV filter image.
- 3- The composed (mask with the cropped image and the red circle).
- 4- The real image without any filter.

### **Move the robot based on the position of the centroid**

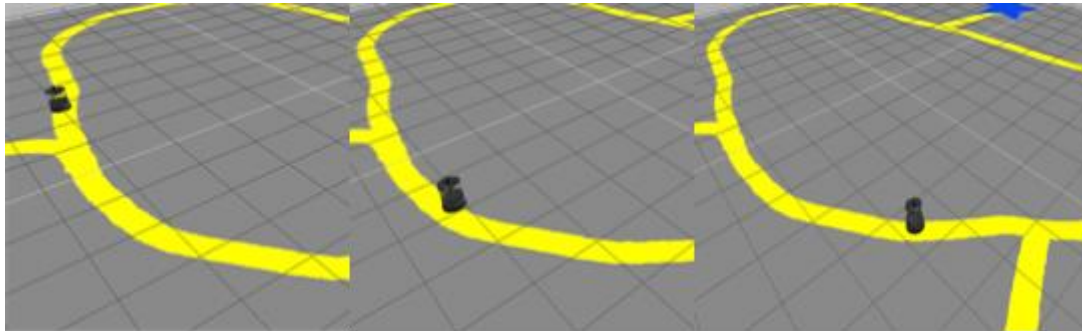
The last step is to move the robot according to all the previous actions. The control of the movement is passed on a Proportional control. That maybe had a small error, but it is one of the best ways to make a line follower.

The idea of the code is to always give a constant linear motion if the camera detected any yellow line and gives the angular Z velocity that depends on the distance between the center of the centroid point of the detected line (the drawn red circle) and the

center of the image then print 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.

We composed all the code in one python file that composed into two classes (MoveKobuki and LineFollower) and the main function. The next figure shows the line follower code's final results that make the robot follow the yellow line using the robot camera.



## Task#2

- 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.

Packages used:

- The camera package [1]
- The cmd\_vel package [2]
- The vision\_opencv package [3]
- The cv\_bridge [4]

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

### Map generation

For the map generation we need a master-object-spawn file to generate the required objects for the map. The objects that need to spawn are depicted in the figure below.

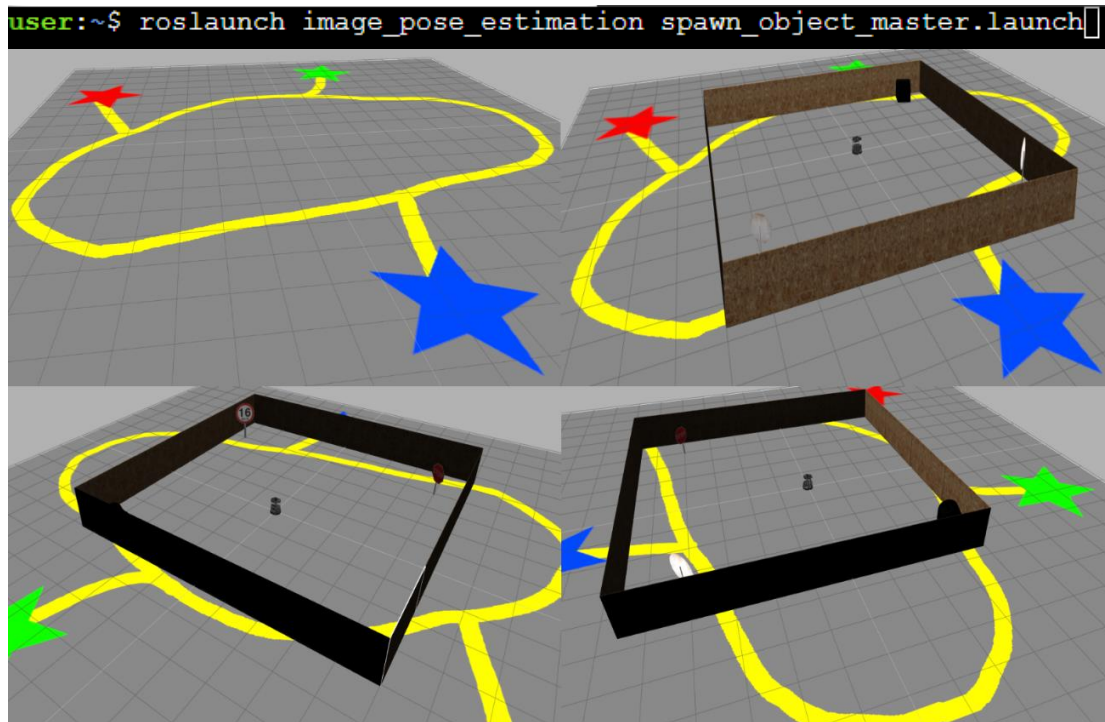
#### Objects



This process is established by launching in the command:

```
< roslaunch image_pose_estimation spawn_object_master.launch >
```

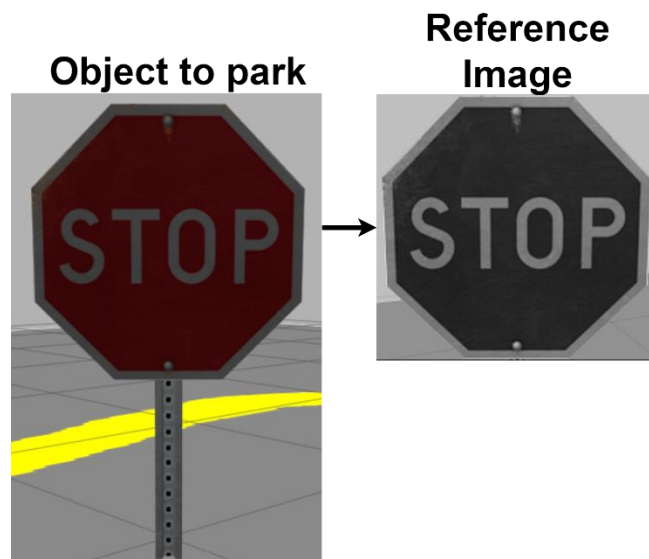




After executing this command, the map is generated, and the robot is ready to start the target initialization task.

### **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 `<image_pose_estimation.launch>` which passes the robot's camera parameters, the blur threshold and the reference image path to the python's variable environment.

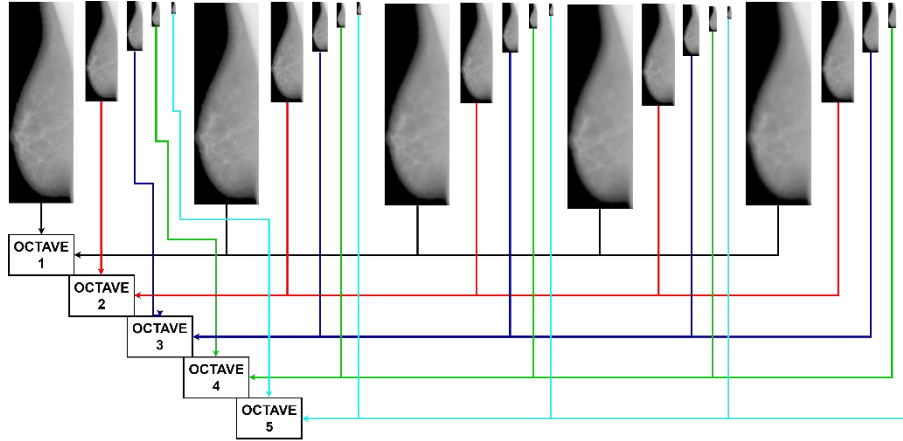
Then features from the reference image are generated using the SIFT [5][6] algorithm for the purpose of having reference features for later matching.

## **SIFT**

The scale-invariant feature transform (SIFT) constitutes a feature detection algorithm used in computer vision for detecting and describing local features in images. Many algorithms have been presented to extract significant features from an image. Some of them use the Laplacian of Gaussian (LoG) to extract these features but they lack accuracy when the image is scaled. In this section we will briefly introduce the crucial steps of this algorithm and how the features are generated.

### ***Step1. Scale space extrema detection***

Unlike other rotation-invariant feature detection algorithms, such as Harris corner detector [7], SIFT uses difference-of-Gaussians (DoG) in different scales as an approximation of LoG (Laplacian of Gaussians) to locate interesting points which scale and orientations do not affect them. To begin with, progressively blurred images ( $L(x, y, \sigma)$ ) are generated by convolving the original image ( $I(x, y)$ ) with the Gaussian kernel ( $G(x, y, \sigma)$ ). Mathematically, this procedure is described as,  $L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$ , where ( $\sigma$ ) is the standard deviation or the “blur” value and  $G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}$ . Then, the original image is resized to half size where a new **octave** (octave is the set of images generated by progressively blurring out an image by altering the blur value) is generated. And this process is repeated analogously with the original size of the image. This procedure is illustrated below .

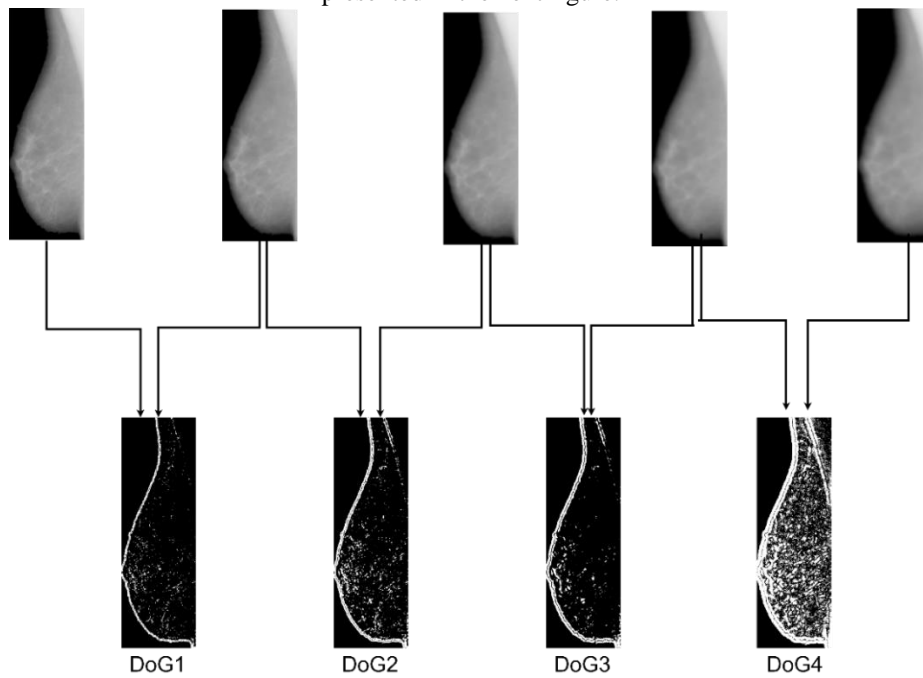


In this figure, each image with same size belongs to an octave which there is progressively blurred (from left to right).

After that, the blurred images are used to generate the DoG set of images for each scale. These set of images, are useful for finding interesting key points in the image. It is not worthy to mention, that DoG images approximate the LoG images and their great advantage is the lower computational cost. More analytically, DoG is computed by:

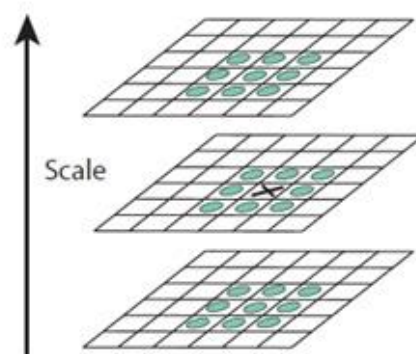
$$\begin{aligned} D(x, y, \sigma) &= (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) \\ &= L(x, y, k\sigma) - L(x, y, \sigma) \end{aligned}$$

An example of the DoG images, or equivalently, the possible interest points from the mias dataset is presented in the next figure.



*In this figure, the (DoG) images are generated, by finding the difference from 2 images at a time. This process is done in all the available octaves.*

Now that the DoG images have been computed, images with less information are produced. The image regions with information contain the “extreme points” which are either the maximum or minimum value points in the image. In more detail, each pixel is being compared by its neighbors. The check is done within the current image, and the one above and below it. One point is larger or smaller than all of its neighbors, then this point is selected as the extrema (candidate).



*In this figure, the marked pixel 'X' is compared with its neighbors from its image, the one above it and one below it (image from [8]).*

### **Step2.Keypoint localization**

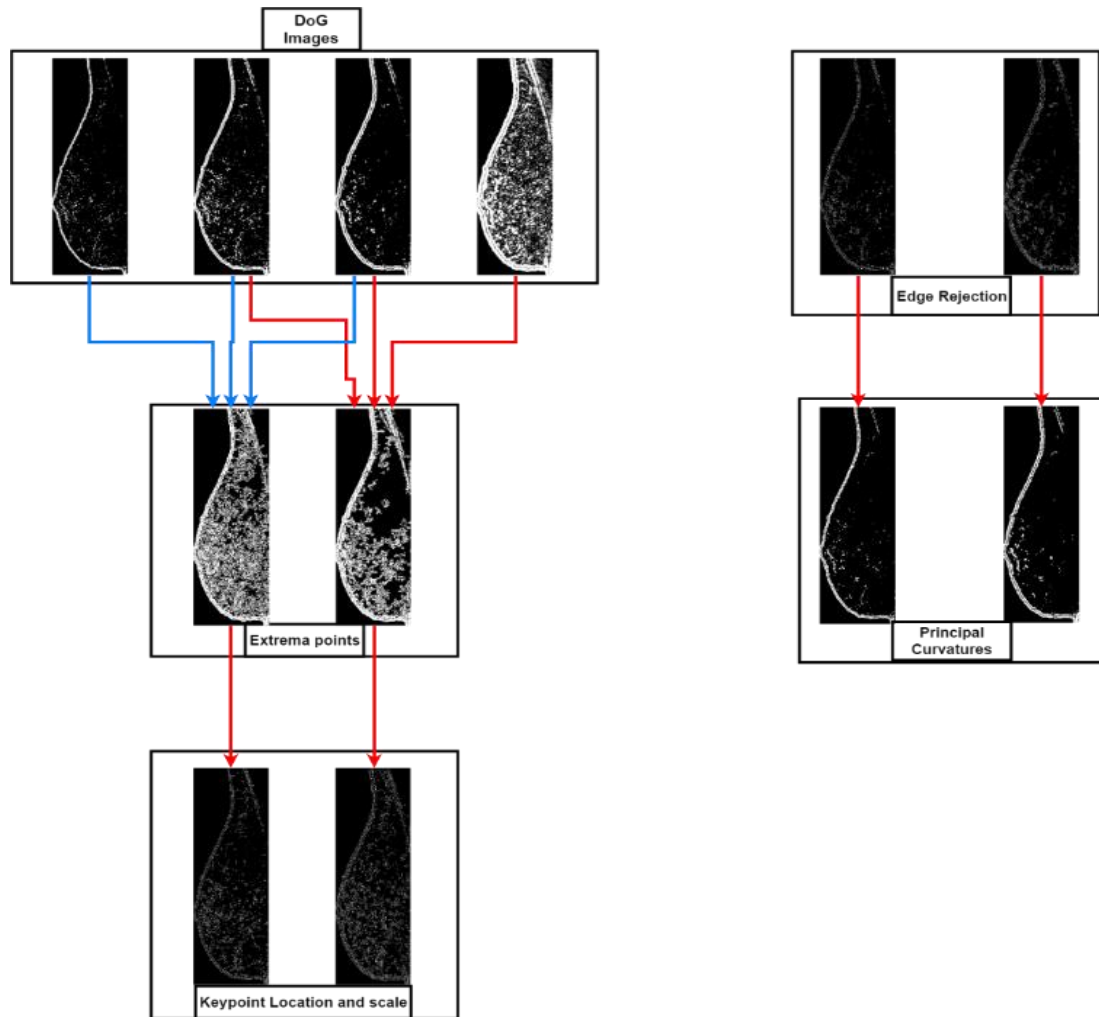
Once a candidate is found, the goal now is to find efficient key points. A selection is made by discarding those points that have low contrast (and are sensitive to light) and the ones that are poorly localized along an edge. The location and scale are calculated using the Taylor expansion of the image.

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}} \mathbf{x} + \frac{1}{2} \mathbf{x}^T \frac{\partial^2 D}{\partial \mathbf{x}^2} \mathbf{x}$$

After that, the **edge** rejection follows, in order to find the principal curvatures. The principal curvatures can be computed from a 2x2 Hessian matrix, H, computed at the location and scale of the key point:

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$$

The result is shown below.

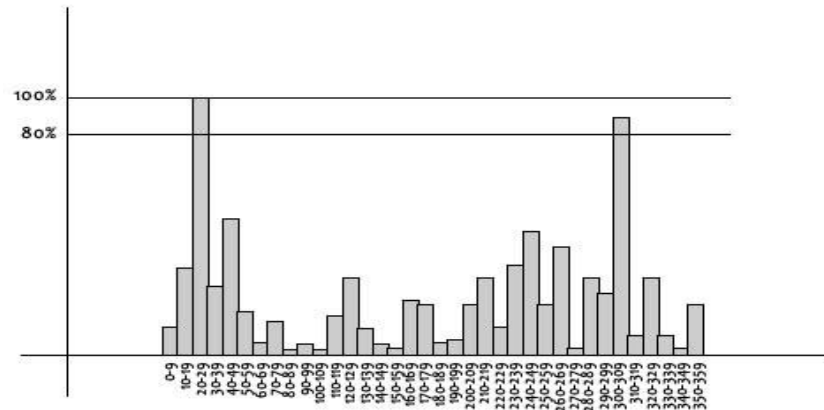


*In this figure, using 3 by 3 images from the DoG set, the extreme points are detected. Then using the Taylor expansion, we find the Location and Scale of Key points in each image. After that, in order to find principal curvatures, edge rejection takes place with Hessian matrix. The result is shown on the set with name “Principal Curvatures”.*

### **Step3.Orientation Assignment**

The next step is to set orientation to the localized key points in order to maintain the image intact after any possible rotation. Around each key point a small region is defined according to scale, orientation and gradient magnitude are calculated separately for each key point region. A bar chart of orientations is created setting 36 equal subsets adding up to 360 degrees representing the, previously calculated, key point neighborhood direction. The key

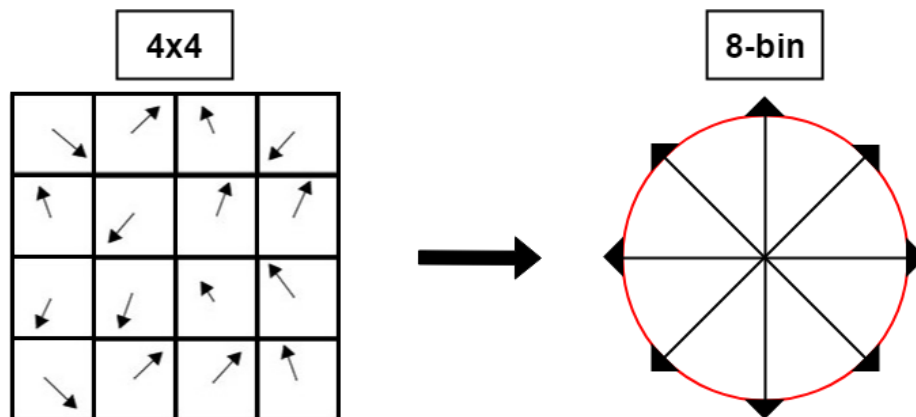
point orientation is estimated by taking the vertex of the histogram and any peak above 80% of it. New key points emerge with same location and scale but different directions.



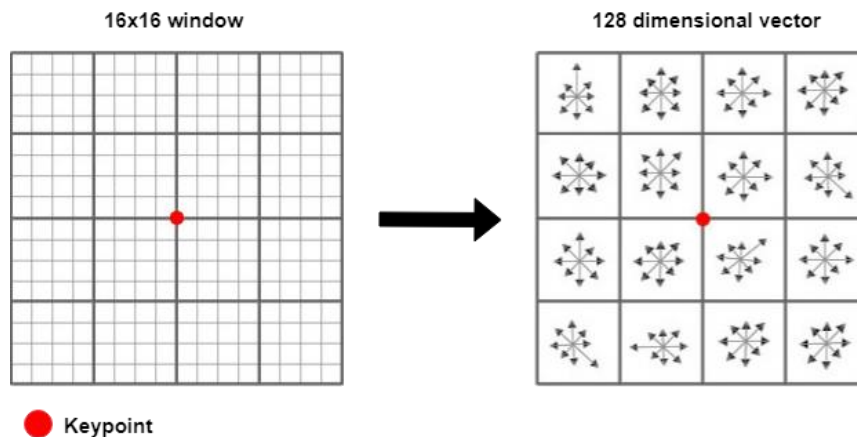
*This histogram represents the degrees in 36 subsets. Each subset is increased by the presence frequency of a degree that a subset has.*

#### **Step 4. Key point Descriptor**

After the generation of key point descriptor, an area of 16x16 blocks around the key point is selected. The division of this area is resulted into 16 sub-blocks with size of 4x4. An 8-bin histogram of orientations is generated for each sub-block. To this end, a sum of 128 bin values is created.

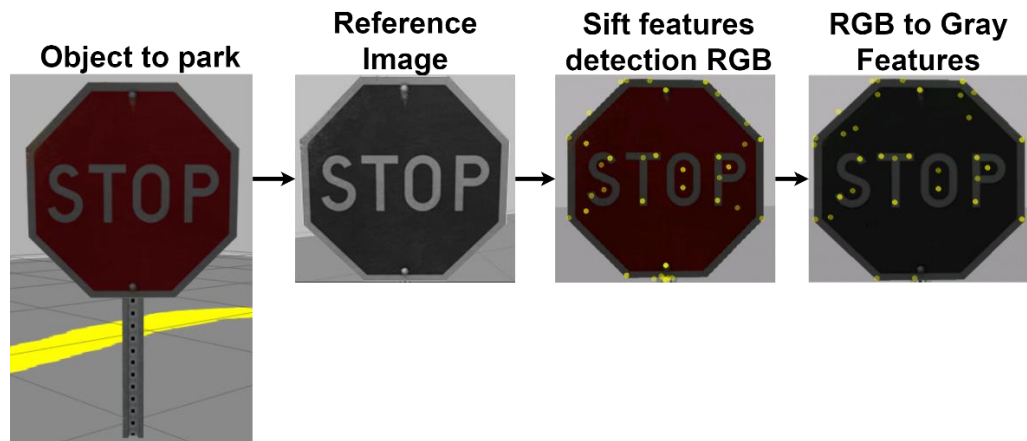


*In this figure each 4x4 sub-block's direction values are inserted into 8-bin orientation histogram.*



*In this figure we see the whole migration of the degree values into 16 8bin orientation histograms.*

Thus, the features of the referenced image are extracted in the figure below.



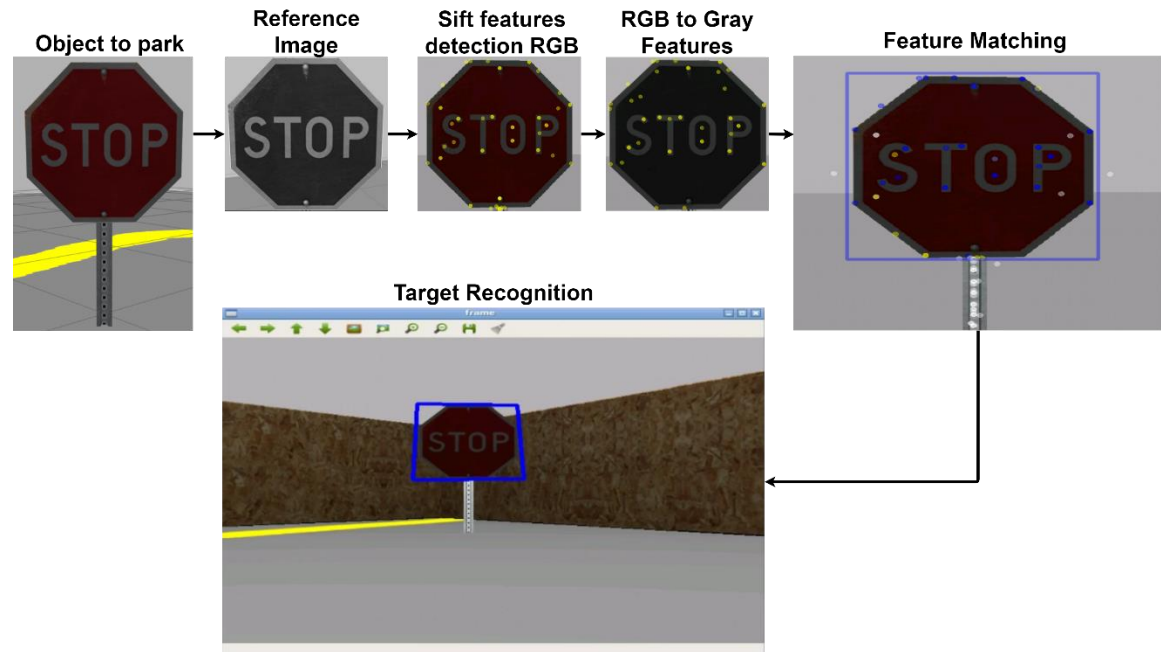
After extracting the features, the robot starts to rotate according to the z axis and starts extracting real-time features from the frames of the camera.





## 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.

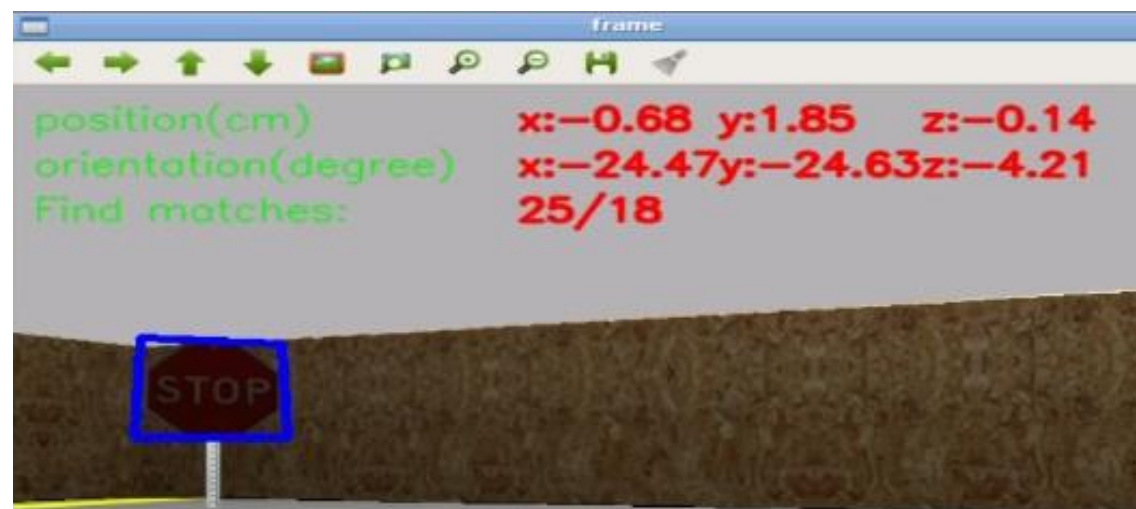


The recognition mode refers to the pose estimation of the object with respect of the rectbox's corners and Homography matrix.

## **Homography**

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 [9].

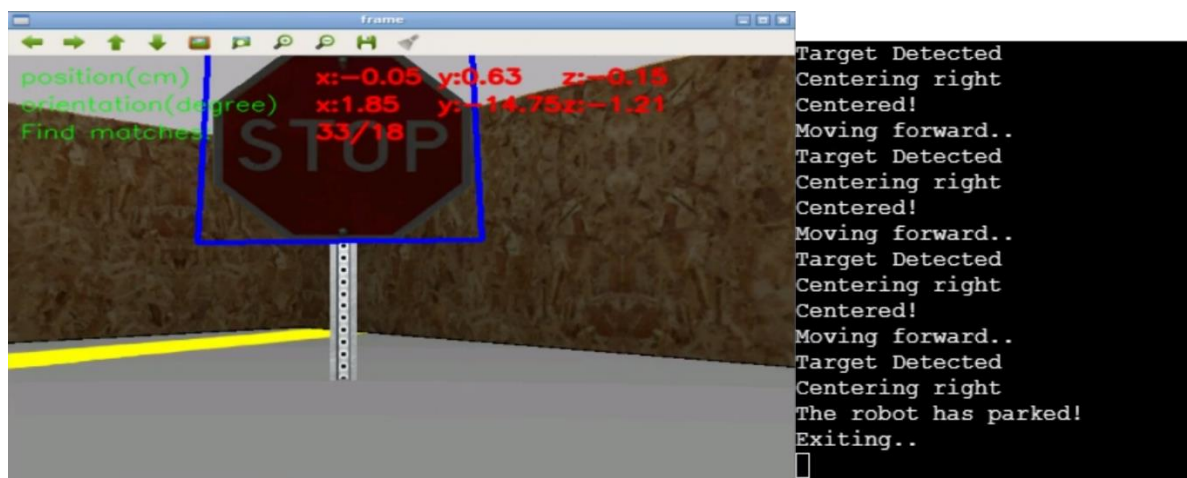
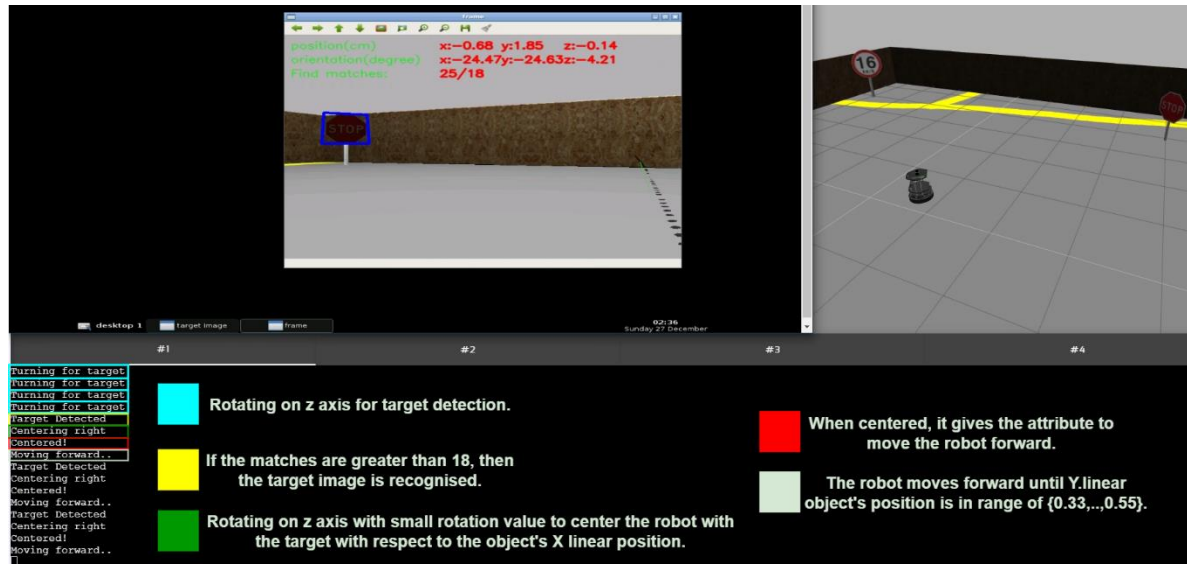
$$\begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} * \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} = H * \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix}$$



*In this figure, the estimated coordinated of the object is shown. The position, orientation and matches amount is depicted.*

## 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."





## 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 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](http://wiki.ros.org/cmd_vel_mux).
- [3] "vision\_opencv." [Online]. Available: [http://wiki.ros.org/vision\\_opencv](http://wiki.ros.org/vision_opencv).
- [4] "cv\_bridge." [Online]. Available: [http://wiki.ros.org/cv\\_bridge](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] C. Harris and M. Stephens, "A Combined Corner and Edge Detector," pp. 23.1-23.6, 2013.
- [8] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-Up Robust Features (SURF)," *Comput. Vis. Image Underst.*, vol. 110, no. 3, pp. 346–359, 2008.
- [9] Abhinav Peri, "Using Homography for Pose Estimation in OpenCV." [Online]. Available: <https://medium.com/analytics-vidhya/using-homography-for-pose-estimation-in-opencv-a7215f260fdd>.