Visual Servoing Project Report

Yağmur Çiğdem Aktaş

November 29, 2021

Contents

Abstract

Path finding and navigation is a frequently encountered task in the field of robotics. Although there are many different methods of transporting a mobile robot from its starting position to a specific destination, in this project, we tried to fulfill this task by using a hand to eye visual servoing system.

Introduction

We carried out a project that processes the pictures taken in real time with a fish eye camera at the top of the environment, calculated the positions of the robot and the target point according to the camera coordinate system, and enabled the robot to reach its target position from this starting position via the shortest path possible. Additional steps such as obstacle avoidance and parking after reaching the target are also among the contents of the project. This project was first prepared on a simulation system and then tested in real time.

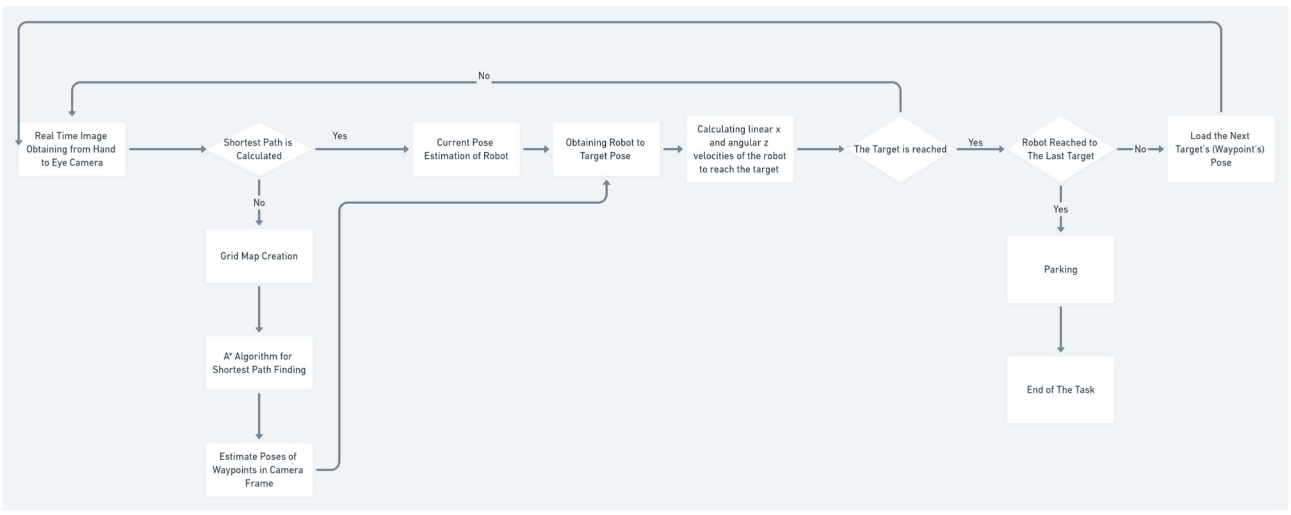
Tools

The hardware and the software tools used for this project:

- Turtlebot3 Burger
- FE185C057HA-1 Fish eye Camera
- Ubuntu 20.4
- ROS Noetic

Tasks and Their Implementation

We tried to implement this project, which is full of different tasks and challenges, by dividing it into simple and small steps. The flow chart below shows the general steps and operation of the project. Therefore, we regularly receive information from our fish-eye camera, which overlooks the environment, and use this information to calculate the starting position of the robot and the target, only once at the beginning. By transforming this calculation into a maze map on which we can use the A* algorithm, we calculate the shortest path between the robot's starting position and the target point, taking into account the obstacles. The way points we obtain as a result of the A* algorithm become intermediate target points that the robot must reach one by one during the stage of reaching the target point. In order to ensure correct speed control of the robot in the linear x and angular z directions, the poses of the way points are calculated according to the camera coordinate system. In another words, we go from image plane (pixel coordinates of the way points) to the real world frame.
After this calculation, the following cycle continues until the robot reaches the target point: Calculating the current position of the robot first in the camera coordinate system and then in the coordinate system of the way point to be reached. Calculation of linear x and angular z speeds using PID controller. When the distance between the current position of the robot and the way point position is less than 0.05 m, the required pose calculations for the next way point are made by sending the information that the relevant way point has been reached. When the robot reaches the target point, parking as the last task is performed by orienting the robot with the same direction of the target pose.


Real Time Image Obtaining from Hand to Eye Camera
Shortest Path is Calculated
Yes
No
Current Pose Estimation of Robot
Obtaining Robot to Target Pose
Calculating linear x and angular z velocities of the robot to reach the target
The Target is reached
No
Yes
Robot Reached to The Last Target
No
Load the Next Target's (Waypoint's) Pose
Yes
Grid Map Creation
A* Algorithm for Shortest Path Finding
Estimate Poses of Waypoints in Camera Frame
Parking
End of The Task

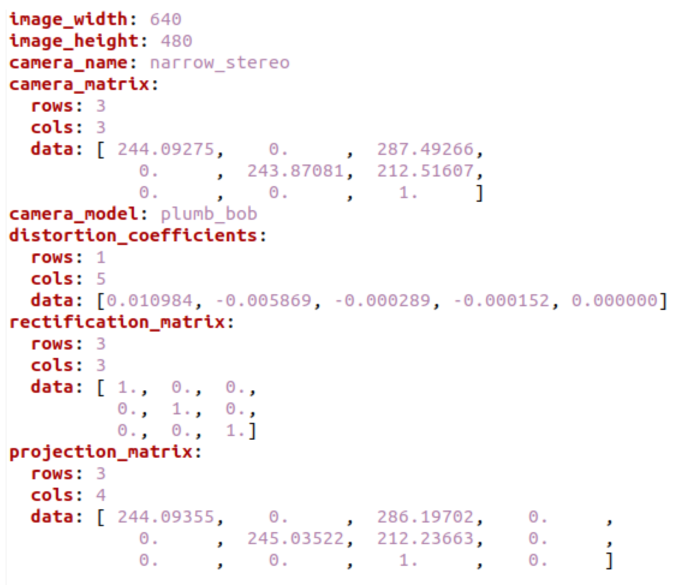
- Calibration of the Camera

  To be able to use the images coming from the camera, we need to undistort them. For this purpose we calibrated the camera using a checkerboard and obtained the camera intrinsic, extrinsic parameters with distortion coefficients.

  We used ROS camera_calibration package for this step with following command:
  rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.08 image:=/camera/image_raw camera:=/camera

  The main link we used is
  http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration

  Finally, we obtained the following "ost_real.yaml" file obtaining camera parameters.



```
image_width: 640
image_height: 480
camera_name: narrow_stereo
camera_matrix:
  rows: 3
  cols: 3
  data: [ 244.09275,    0.     ,  287.49266,
            0.     ,  243.87081,  212.51607,
            0.     ,    0.     ,    1.     ]
camera_model: plumb_bob
distortion_coefficients:
  rows: 1
  cols: 5
  data: [0.010984, -0.005869, -0.000289, -0.000152, 0.000000]
rectification_matrix:
  rows: 3
  cols: 3
  data: [ 1.,  0.,  0.,
          0.,  1.,  0.,
          0.,  0.,  1.]
projection_matrix:
  rows: 3
  cols: 4
  data: [ 244.09355,    0.     ,  286.19702,    0.     ,
            0.     ,  245.03522,  212.23663,    0.     ,
            0.     ,    0.     ,    1.     ,    0.     ]
```

- Real Time Image Handling

  Using the fish eye camera on the top of the environment, we obtain the first image to extract initial robot position and target position using the Aruco Markers having 15x15 cm size. For this purpose we have a subscriber which listens to the camera/image_raw topic. In the callback function of this subscriber, we detect the 4 corner of the markers as the pixel coordinates in the image. This first image and initial pixel positions are used for map creation in the next step.

Apart from the first picture sent for mapping and the initial robot - target detection, this step is responsible for publishing the current robot position to the estimated_pose topic through a publisher, by first detecting the current position of the robot on the picture and then calculating its position according to the camera coordinate system. This step is performed repeatedly until the whole task is completed.

This task is implemented in **real_time_image_subscriber.py** file. Main parts and their explanations of this file is as follows:

sub = rospy.Subscriber('/camera/image_raw', Image, callback) → sub is the main subscriber to the camera/image_raw topic where our fish eye camera publishes the images

callback(msg) function → callback function of the sub subscriber. In this function, the image is converted to a numpy array, undistorted with **cv2.undistort()** function using calibration parameters. After undistorting, the markers are detected using **cv2.aruco.detectMarkers()** function which gives us the id and 4 corners of a detected marker. Using **cv2.aruco.estimatePoseSingleMarkers()** function, these 4 corners are used to calculate the camera frame positions of the markers. This step is the main point to keep tracking the robot's current position.

pub = rospy.Publisher('/estimated_pose', MarkerPose, queue_size=1) → pub is the main publisher variable to send the current pose of the robot to the estimated_pose topic using MarkerPose user defined message type. This message contains rotational and translational vectors of the pose.

- Map Creation, Shortest Path Finding with A* Algorithm & Pose Estimation of each Way Point

The first image obtained by the camera is saved as "env.png" in real_time_image_subscriber.py file.

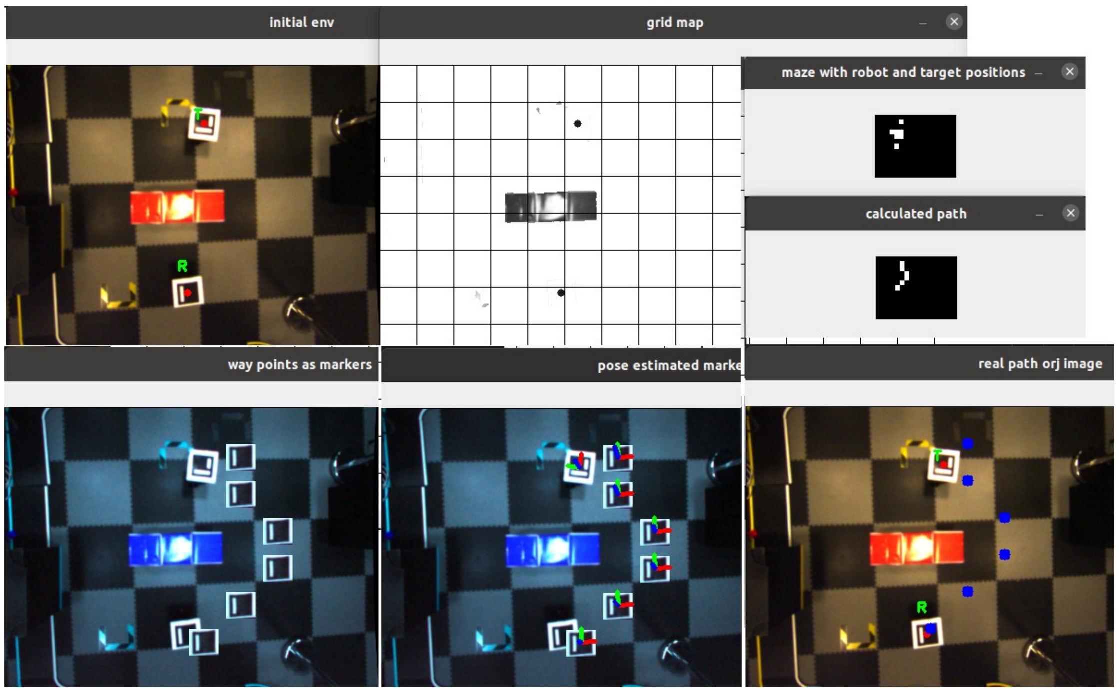
1. At first, robot's and target's marker are detected. Since the robot's marker has the ID 0 and target's marker has the ID 1, we can easily distinguish which is the robot's marker detection and which is the target's marker detection.

2. Secondly, we detect red boxes applying an RGB low – high value mask to our image. After applying this mask, we obtain an image where only the obstacles, robot's and target's center points are visible while other pixels are all black.

3. Next step is to create a grid map using these obstacle – non obstacle pixel information. A grid map of this image is created using 40x40 pixels of grid boxes. To obtain a binary image named maze having 0 for free spaces and 1 for obstacles, we travel through our initial image with a 40x40 window size, we put 1 if any obstacle pixel is found in this window area and 0 otherwise.

4. After obtaining the maze, we can directly apply A* star algorithm giving this maze, robot's position as start point and target's position as end point. To convert robot and target pixel positions to the maze pixels, we divide these pixel coordinates by 40 again.

5. A* algorithm gives us the way points and the last step of this part is to convert these way points to the camera frame positions which is done by using cv2.aruco.estimatePoseSingleMarkers function again. To be able to use this function we treat the way points as marker points by copying the marker size images to the initial image. These way point positions will then be used in controller to calculate the robot speed to reach the way points one by one until the main target is reached.



map() **function of** **create_map.py** file is the main module which applies these steps and return a python list containing homogenous matrices of each target point where a homogenous matrix of a detected pose is as follows:

$$\begin{bmatrix} r11 & r12 & r13 & t1 \\ r21 & r22 & r23 & t2 \\ r31 & r32 & r33 & t3 \end{bmatrix}$$

- Calculating Robot Pose in Way Point Coordinate Frame & Navigation of the Robot

After finding the path to follow, we take the current position of the robot coming from real_time_image_subscriber.py and the first way point position to calculate the robot position in the target frame. That gives us the difference in x and y axes as well as the rotation in z axes which are the all we need to calculate the linear x and angular z speed of the robot in order to move through the target direction.

This calculation is made by multiplication the **inverse of the homogeneous robot position matrix by the homogeneous target matrix**.

$$\begin{bmatrix} \begin{matrix} R \\ (3X3) \end{matrix} & \begin{matrix} deltax \\ deltay \\ delta\Theta \end{matrix} \end{bmatrix} = \begin{bmatrix} r11 & r12 & r13 & t1 \\ r21 & r22 & r23 & t2 \\ r31 & r32 & r33 & t3 \end{bmatrix}^{-1} * \begin{bmatrix} r11 & r12 & r13 & t1 \\ r21 & r22 & r23 & t2 \\ r31 & r32 & r33 & t3 \end{bmatrix}$$

Using **arctan2(deltay / deltax)**, we calculate the orientation angle that the robot should take, we wait until the robot fixes its angle until the error goes lower than 5 degrees, then let it go straight forward on x axis direction.

During the orientation step, we calculate the angular z axis speed velocity of robot using **w = k_alpha * alfa** equation which gives us the velocity in **radians**. Here, alfa is the result comes from alfa = np.arctan2(deltay, deltax) and k_alpha is a constant we determined as 1. During this step, linear x velocity is 0 since we are waiting the robot to fix its orientation before start moving towards the target.

Once the orientation angle is fixed, we calculate the linear x speed to move the robot forward. This velocity v is calculated using **v = k_p * p** where p is the euclidian distance between robot and target pose and k_p is a constant we determined as 0.8. To calculate the euclidian distance we use deltax and deltay **p = math.sqrt(deltax**2 + deltay**2).** The threshold to decide if the robot is reached the target is 0.045 meter.

In general, if the robot fixes the orientation angle, linear x speed is set to 0 and if the robot moves straight forward, the angular z speed is set to 0. We check the orientation error after the initial orientation, while the robot goes forward, and if the error starts to increase again, we stop the robot's forward speed and let it fix the orientation again. For that step, the threshold for the orientation error is 15 degrees.

This process is running in a loop for each real time image obtained by the subscriber until the robot reaches every way point and the target position at the end.

This task is implemented in **controller.py** file and here is the main parts and explanations of this module:

pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1) → main publisher to send the calculated speed to the Turtlebot3

sub = rospy.Subscriber('/estimated_pose', MarkerPose, callback) → main subscriber to obtain the current robot pose published from real_time_image_subscriber.py module

target_matrices = create_map.map() → Poses of the way points and final target point calculated in the previous step are stable and they are not calculated after first initialization. These poses are stored in this Python list variable.

callback(msg) → The callback function of subscriber. Each time a new current pose of the robot is sent by real_time_image_subscriber, this callback function calls controller function with the related target homogeneous matrix and robot's.

controller(homogenous_robot,homogenous_target) → Robot's pose in target frame is calculated in this function. After that, according to the mode (initial orientation, going forward or parking as we will explain in the next part), the speed is calculated as explained above and it is published. **Every time the robot reaches to a target point, this function return True to inform callback function for sending the next target position matrix.**

Since the Turtlebot3's maximum angular velocity is 2.84 radians/sec and maximum linear velocity is 0.22 m/sec, we determined the following constraints to be sure we don't publish any velocity information that may harm the robot.

```
if v > 0.22:
        v = 0.22
elif v < -0.22:
        v = -0.22

if w > 2.84:
        w = 2.84
elif w < -2.84:
        w = -2.84
```

These constraint are applied just before publishing the velocities independently from the mode.

- Parking

After reaching the final target position, the robot is parking according to the rotational direction of the target position. We obtain this information on deltaΘ coming from the robot-to-target homogenous matrix unlike the teta error which was coming from the arctan of deltay and deltax during the path finding. The basic difference between these two angle errors is that during the path finding, we rotate the robot in the target position direction by not taking account in which direction the target positions oriented exactly. In the parking step, we orient the robot to look exactly the same direction with the last target position. For this purpose, we stop the robot’s linear x speed and only adjust angular z speed until the error goes lower than 5 degrees.

parking(homogenous_robot,homogenous_target) → main function for parking step implemented in controller.py module. This function is called after the target_reached flag is set True at the end of the previous step.

Resources & Conclusion

The Construct
Emanuel Robotis
A star algosunu aldığın medium linki