Lab 4: Virtual Servoing

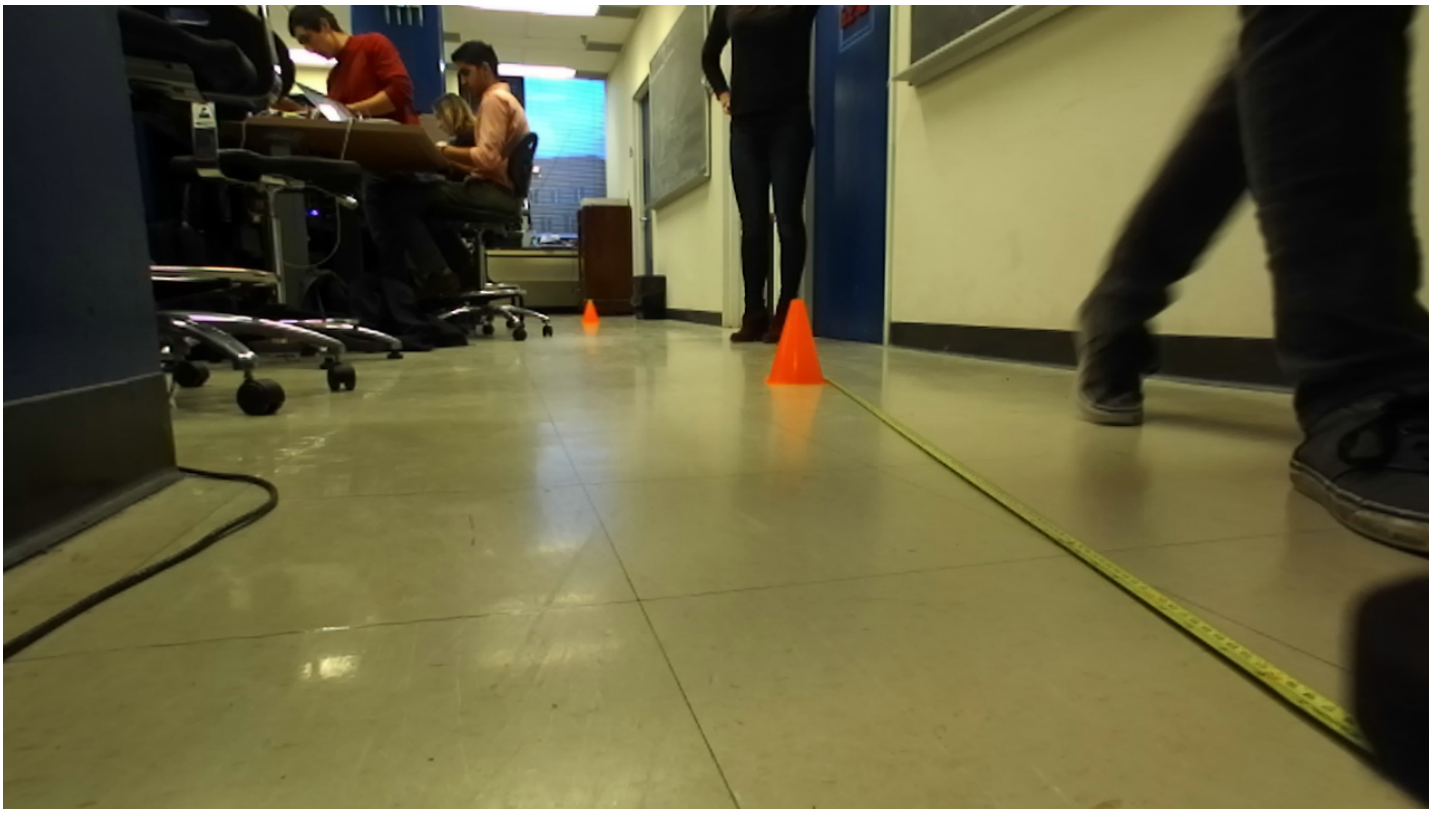
RSS Team 8
Kevin Chan, Alexander Galitskiy, Josh Hellerstein,
Trevor Henderson, Caroline Hope, Katy Muhlrad
6.141/16.405J Robotics: Science and Systems
Spring 2017

Submitted: March 17, 2017

Lab 4: Visual Servoing

Lab 4 is a two-week and two-part lab regarding visual servoing. During Part A of the lab, teams were asked to explore various computer vision object-tracking algorithms (such as SIFT, template matching, or color threshold blob detection) to track a small orange cone. The robot can find a cone in its view and then park itself a set distance in front of the cone, as well as maintain a set distance from the cone as it moves. In Part B of the lab, our group implemented line following algorithms to drive around a circle. We compare results using open-loop control, setpoint control, and pure pursuit control schemes.

Lab 4A: Cone Parking



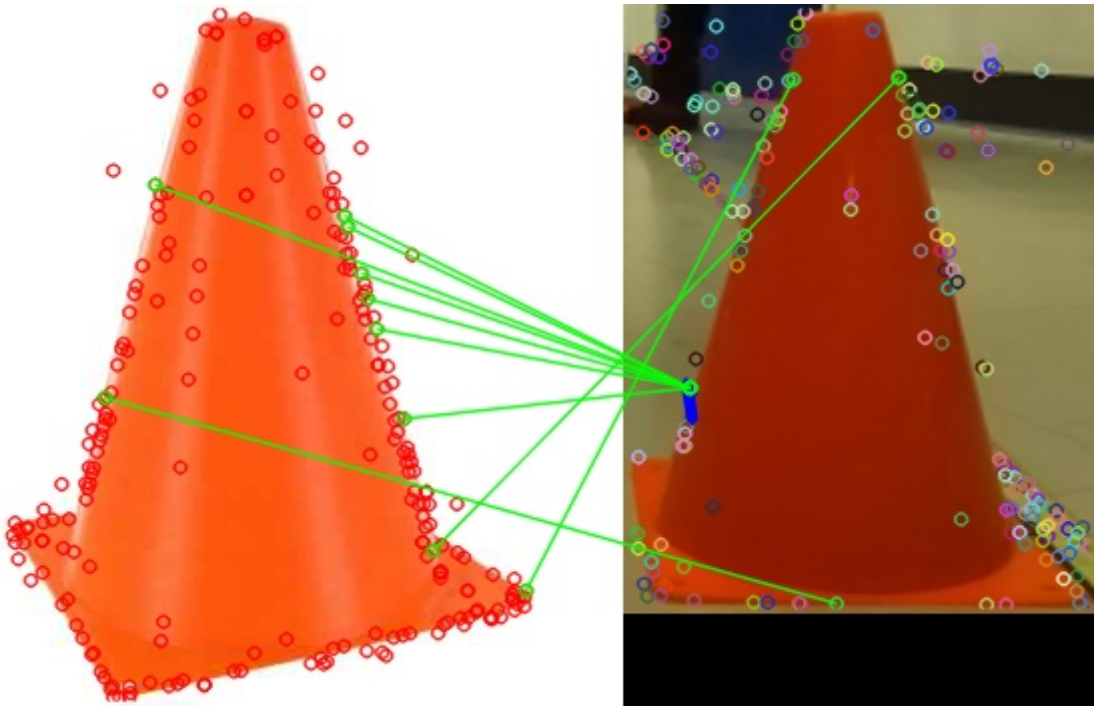
The first part of the lab asks us to park our car 1.5-2 feet from a small orange cone. The image above was captured from the onboard camera while parked 150cm from the cone. The main challenges in this task are cone detection and parking control.
[Kevin]

Cone Detection

The cone detection methods output a bounding box around cones in the image. These bounding boxes can be used to calculate the distance and angle to the cone.

Method 1: SIFT and RANSAC

Initially the team attempted to implement a Scale-Invariant Feature Transform (SIFT) and Random Sample Consensus (RANSAC) algorithm. This object detection method is historically quite robust and is useful for not confusing background with the intended object. This is because the algorithm is intended to find keypoints in the current input image that correspond to features in a reference image, making the method robust to scale, orientation, illumination, and viewpoint [1]. The issue that the team ran into with this method was that the cones do not really have features for the algorithm to sense as keypoints. This means that most of the keypoints found by the algorithm were in the background since the cones were featureless, and the keypoints that were on the cone were replaceable around most points on the reference cone, making the robot think that the image was rotated or did not exist in the frame. [Caroline]

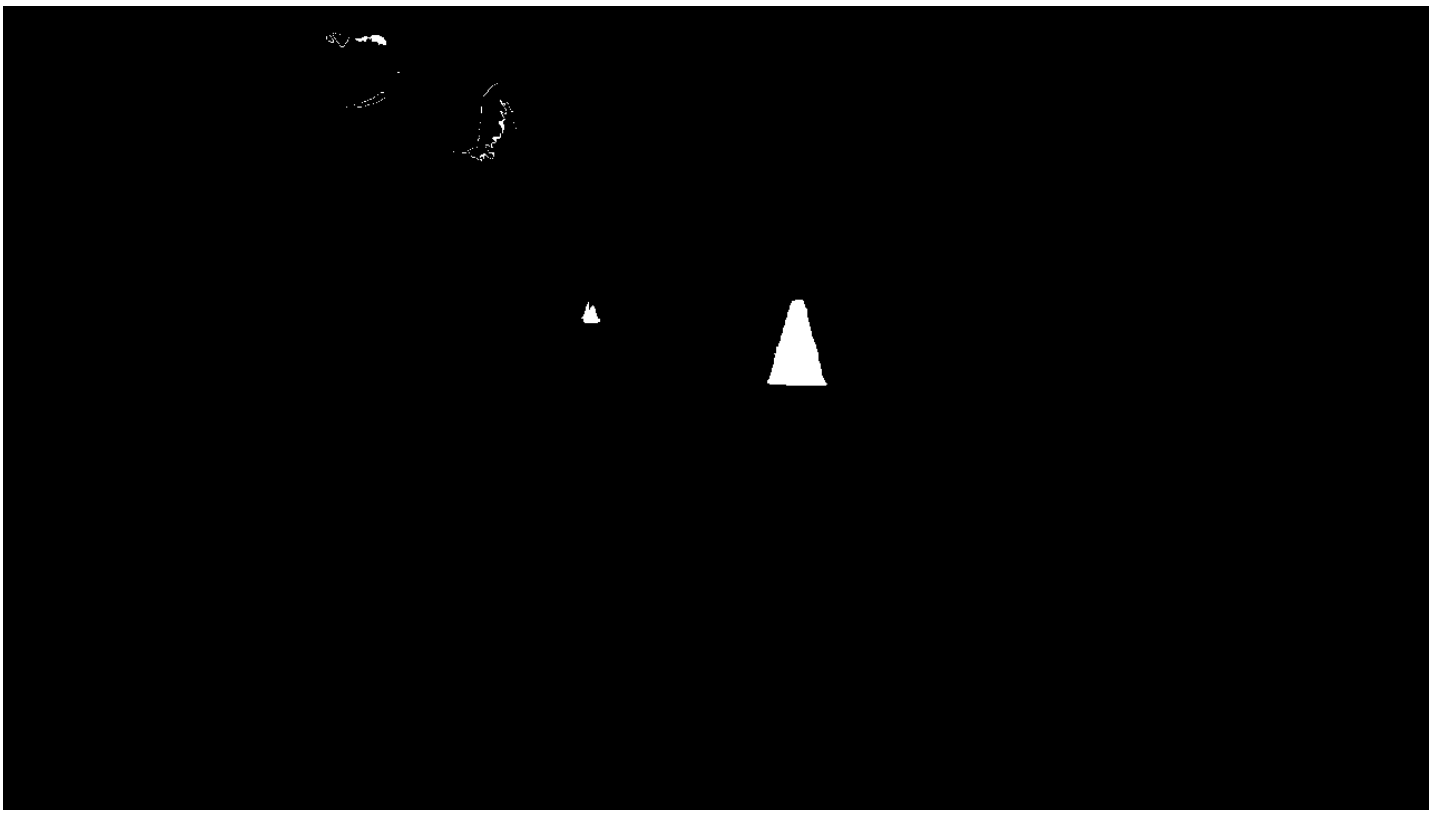


Method 2: HSV Blob Detection

When SIFT didn't work for the cone detection, we decided to try an HSV blob detector. The first step was doing a simple HSV threshold. We converted the images received from the camera into HSV space. Then using threshold values determined experimentally, we filtered the image by color, keeping only spots identified as "orange".

```
# Get the mask of an image to filter out
# everything but orange

self.HSV_MIN = np.array([6, 90, 160])
self.HSV_MAX = np.array([15, 255, 255])

def getMask(self, hsvImage):
    # HSV Threshold
    mask = cv2.inRange(hsvImage, self.HSV_MIN, self.HSV_MAX)
    # Erode and dilate the mask to
    # clean up noise and reconnect points
    mask = cv2.erode(mask, None, iterations = 1)
    mask = cv2.dilate(mask, None, iterations = 3)
    return mask
```

This is an example of the filtered version of the above picture of a cone 150 cm away from our car.



In order to filter out the noise, we employed 3 different filters on: shape matching score, aspect ratio, and area. OpenCV's methods `findContours` and `boundingRect` gave us information about all the objects after applying the color threshold, including their widths, heights, and pixel coordinates.

To get a good idea whether one of the identified objects was a cone, we wanted to compare it to an object we knew was a cone. We resized a sample picture of the cone to the size of the new bounding box. Using OpenCV's `matchShapes` method, the contours of

the images were compared for similarities. The more similar the objects, the lower score was returned. An image would return a score of 0 if matched with itself.

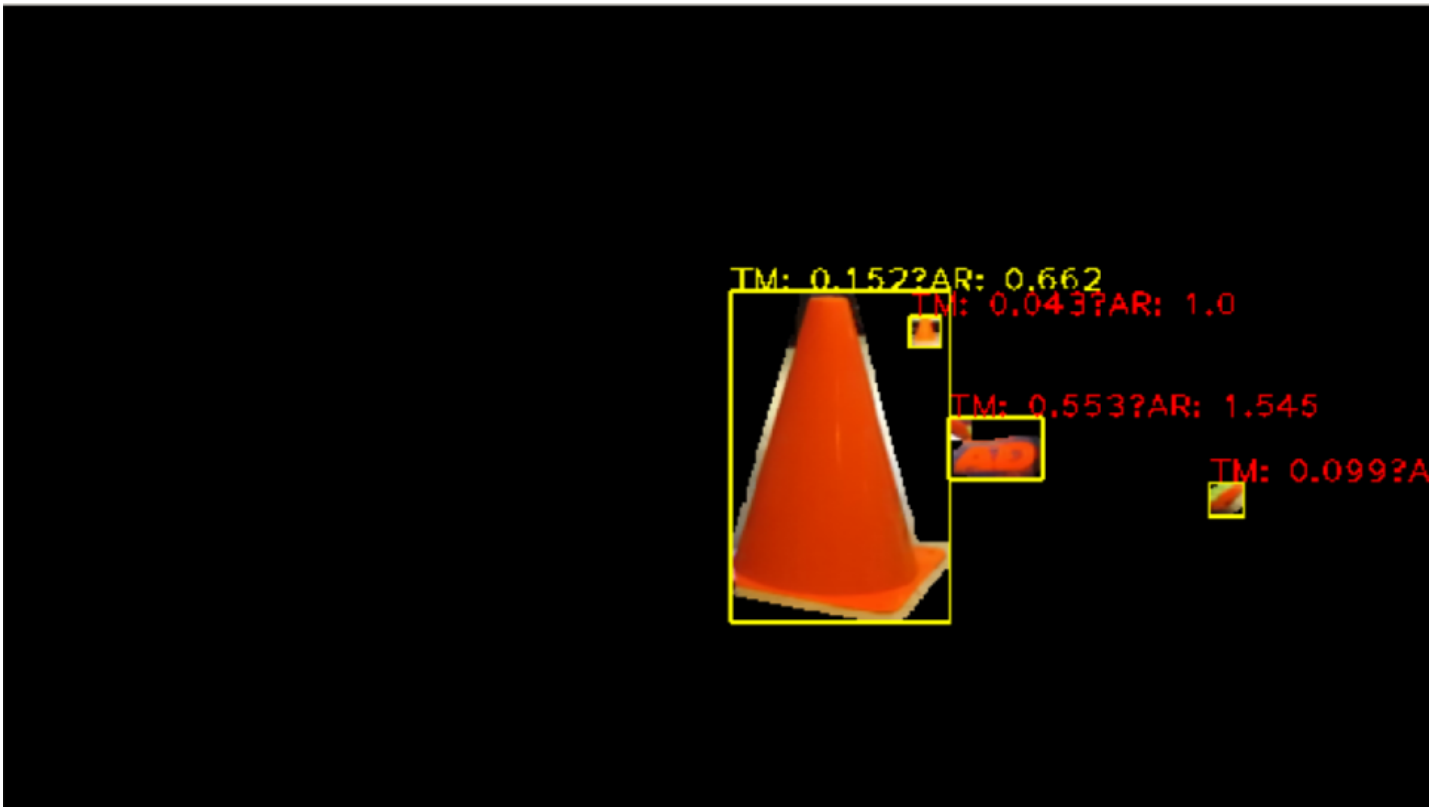
After checking if the objects looked similar to an image of a cone, we wanted to confirm its similarity to an actual cone. By looking at a lot of different sample cone images with the cone at varying angles and distances away from the robot, we found that the aspect ratio of the cone was about 0.65. This filter helped get rid of a lot of noise that was wider than it was tall, even if the noise was slightly cone-shaped.

Finally, if there were two different objects that were both very “cone-like”, we chose the biggest one to target. This meant that if there were two cones within the camera’s field of view, our robot would only consider the closest one. Below is the code where we do the main filtering.

```
# Return the best object out of all identified
# objects based on shape matching score, aspect
# ratio, and area

def filterContours(self):
    matchValues = self.templateMatching()
    self.filteredContours = []
    bestArea = 0
    self.bestWidth = None

    for i in range(len(self.contours)):
        contour = self.contours[i]
        matchValue = matchValues[i]
        xTop, yTop, width, height = cv2.boundingRect(contour)
        aspectRatio = width/float(height)

        if (matchValue < self.MATCH_MAX) and (self.ASPECT_MIN < aspectRatio
            if (self.ASPECT_MIN < aspectRatio < self.ASPECT_MAX):
                self.filteredContours.append(contour)
                area = width * height

                if area  >= bestArea:
                    bestArea = area
                    self.bestContour = contour
                    self.bestWidth = width
                    self.bestHeight = height
                    self.bestxTop = xTop
                    self.bestyTop = yTop
```

The figure below shows the final results of our cone processing. The bounding boxes of all thresholded objects were drawn with their shape matching scores and aspect ratios. The boxes with the red text were all filtered out, so the only object the camera considered

was the large cone with the yellow text. [Katy]



Parking Controller

The objective for the parking controller was to:

1. Have the car park 1.5ft in front of a cone from a starting distance of > 6ft.
2. Have the car maintain a set distance of 1.5ft from a moving cone.

After implementing the computer vision algorithms discussed above, we had to determine the distance and angle of the cone from its pixel coordinates/representation, in order to park the car accordingly. We collected 20 image samples measuring cone-height in pixels (from the bounding box determined using color thresholding), and real-world distance of the cone.

First, we tried to directly plot the relationship between pixel height and known, and regress a 7th degree polynomial to the relationship. This process was complex and still had less precision than wanted. The next attempt to model distance was by using the bounding box pixel height to find an intermediate real-world theta value, and then to distance. However, we soon realized this was insufficient for when the cone was at large angles to the car, due to the properties of the camera.

Our working model is quite simple, and simply relies on the fact that in a pinhole camera, the height of an object is inversely proportional to the distance of the object. We simply had to find a single constant representing the focal length in pixels times the real world height of the object and divide it by the height of the object.

```
DIST_TIMES_FOCAL_LENGTH_PX = 50/0.0068
return self.DIST_TIMES_FOCAL_LENGTH_PX/self.getHeight()
```

We could also determine the angle offset of the cone from the car by determining how far the center of the bounding box of the cone was off from the image center (where the image size was normalized to 1):

```
  return (self.bestxTop + 0.5*self.bestWidth)/self.imageWidth - .5
```

Once we determined the distance and angle offset, we were able to create a proportional-derivative (PD) controller for steering and velocity of the car.

```
steering_angle = -theta - self.ANGLE_KD*(theta - self.last_angle)
error = np.log(angle_range.range/self.SET_RANGE)
```

Here, we take the log of the ratio between the actual distance and the set distance for the cone to the car, to set the error for the velocity computation. We did this because the velocity was too small when the car was close to the cone, and extremely large when far away from the cone.

[Josh]

Cone Parking Results

Using the HSV blob detection with further filters worked really well for consistently identifying the cone and only the cone. However, the environments we were running the car in didn't inherently have a lot of noise, so the extra processing was unnecessary. The shape matching criteria was taken out of the final implementation to help speed up the processing without any penalties on actual performance.

The nonlinear (log) factor scaling the error for velocity control made the car extremely responsive at small distance from the cone, and have a more realistic velocity far distances from the cone, which was a much favorable response to the purely linear controller.

Due to the high resolution (2k) and frame rate (60-100hz) of the Zed camera, we had to decrease the resolution to decrease latency in the pipeline. The higher resolution image took too long to process, and resulted in a 3 second delay. With the resolution decreased to VGA, our car became latency-free.

All together, our robot maintained a steady distance to the cone and parked without oscillations at both close and far distances.

[Katy, Josh]

Lab 4B: Line Follower

For the next part of the lab, teams were asked to extend their cone following implementations to a line follower. Three different controllers of a line follower were implemented: an open-loop controller, a setpoint controller, and a trajectory tracking controller. Though assumed that the trajectory tracking controller would have the best line-following capabilities, all three are implemented in order to compare performance. The performance metrics included the maximum stable speed and number of loops around a circle.
[Caroline]

Open Loop Control

Overview

The open-loop controller only published a constant drive speed and steering angle to the car. After beginning with a slow speed of 0.1 m/s and small steering angle of less than 15 degrees, both of these parameters were changed to attempt to stay on the circular path for as long as possible. The steering angle was decreased to fit the circle with a 5’ radius marked on the floor by the course staff. The speed was slightly increased for testing.
[Caroline]

Testing

Testing of the open-loop controller involved calibrating the steering angle until it started to follow the circle for a short time. We had started at just less than 15 degrees (`STEER_ANGLE = 0.2 rad`) as an estimate. This value for steering angle still turned too much so we calibrated and eventually came to a value of `STEER_ANGLE = 0.13` that followed the circle. We increased the speed slightly but did not attempt to test very fast

speeds because even with the calibrated steering angle the car diverged from the circle at after only completing about ¼ of the way around, as stated in Table 1 below.
[Caroline]

Set Point Control

Overview

The setpoint controller utilizes the same HSV color blob threshold as the initial cone detector. The first change from part 4A of the lab is the the upper three-fourths of the input image are masked so that controller only samples from the bottom fourth of the image. It also did not apply the bounding box based on any geometric parameters like aspect ratio or shape matching, only based on color. This assumes that there will not be any background noise to confuse the HSV color blob thresholding. This is a good assumption since the only part of the image being processed displays only what is closest to the robot (about 10 cm to 25 cm in range).

The controller then took the location of the bounding box and calculated the horizontal error from the center point of the tracked orange tape to the center point of the Zed Camera image. This error was converted to a normalized location horizontally on the image and then into radians to be used for steering angle control. For simplicity, the controller was scaled by a proportional term but did not utilize a derivative term or an integral term.
[Caroline]

Testing

The setpoint controller worked quite well on the 5ft circle at 0.466 m/s, and most likely could have been run at a faster speed. It was tracking to the outside of the circle (left wheels just inside the circle), which is thought to occur because the input images were coming from the left Zed camera. This would cause robot to align to the center point of the left camera instead of an image from the center of the robot.

The robot was also tested on the tape path in the lab that turns in both directions. The robot was able to follow the line well for the shallow turns but ended up turning off the path for the stronger turns. The proportional gain was increased and decreased to see the effect. Increased gain allowed it to react greater when it sensed the turns, but we saw greatly increased oscillation and the robot still was not able to make it around the turn well. The decreased gain got rid of the oscillations but none of the attempts let the robot make it around the turns.
[Caroline]

Trajectory Tracking

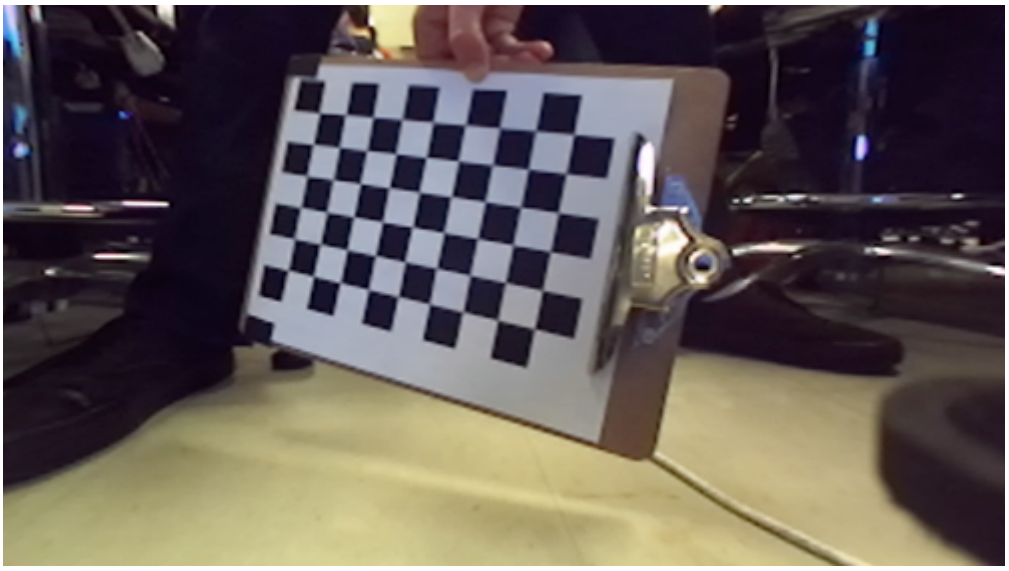
Overview

We next implemented a Pure Pursuit trajectory tracking algorithm to drive the circle as fast as we could.

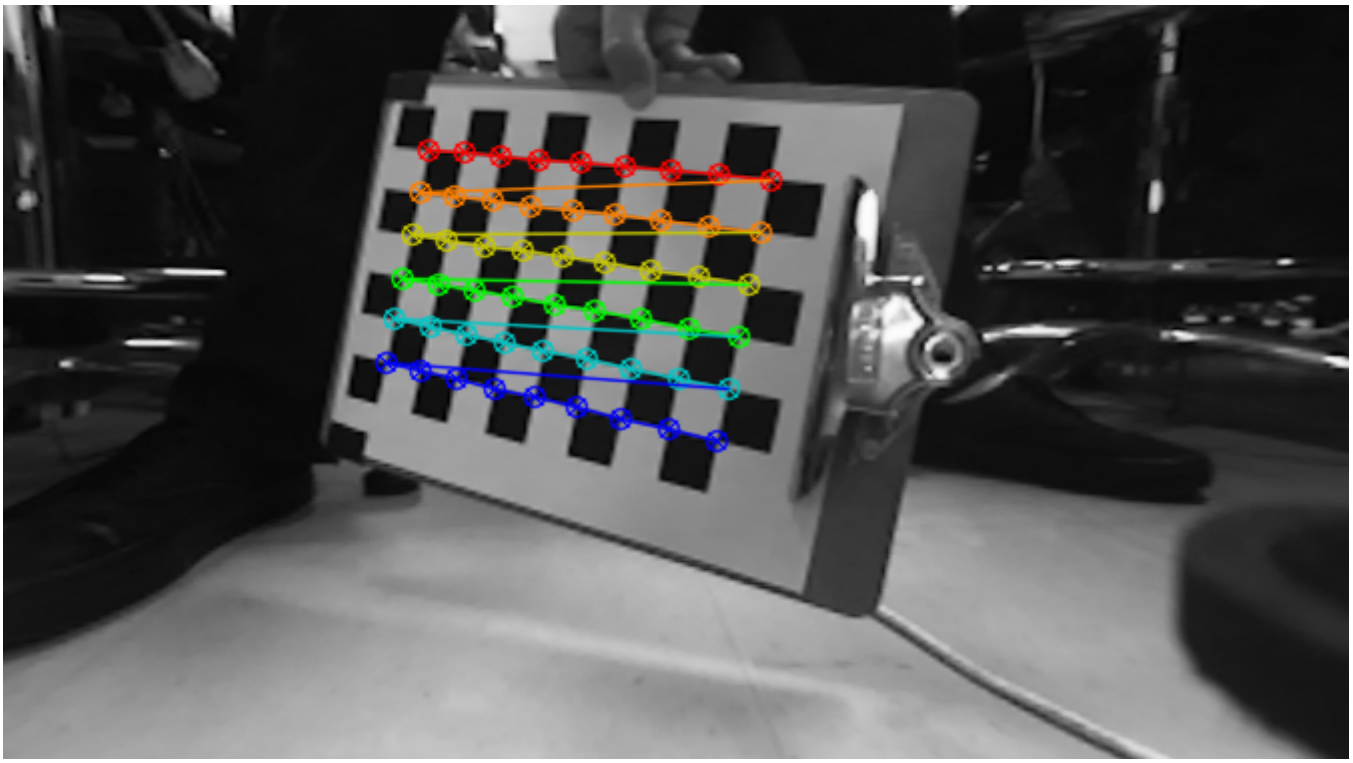
Camera Calibration

In order to properly implement trajectory tracking, we first need to convert from “pixel coordinates” to “world coordinates”. Pixel coordinates refer to pixel indices in the image taken from the onboard camera, we will call (u, v). We define our world coordinates on a plane along the ground with the origin at the center of the rear axle of the car and to have units of centimeters.

We first found the camera matrix of the ZED camera, which encodes information about the curvature of the lens. To do this we took about 20 photos of a 10 by 7 checkerboard as picured below.



We used OpenCV’s `findChessboardCorners` function to get the corner points of this checkerboard and the `cameraCalibration` function to turn this set of corner points into a camera matrix. [2]



The camera matrix we found was

$$C = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 351.7 & 0 & 306.25 \\ 0 & 353.7 & 183.9 \\ 0 & 0 & 1 \end{bmatrix}$$

where are the f_x and f_y are the focal lengths and c_x and c_y are the optical centers of the camera expressed in pixels for an image of size 672×367 pixels.

The `cameraCalibration` function also returned the rotation matrix R and translation vector $\mathbf{t}$ of each checkerboard relative to the camera. Therefore for affine world coordinates $\mathbf{x}$ the image coordinates were as follows

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = C \, [R|\mathbf{t}] \, \mathbf{x}$$

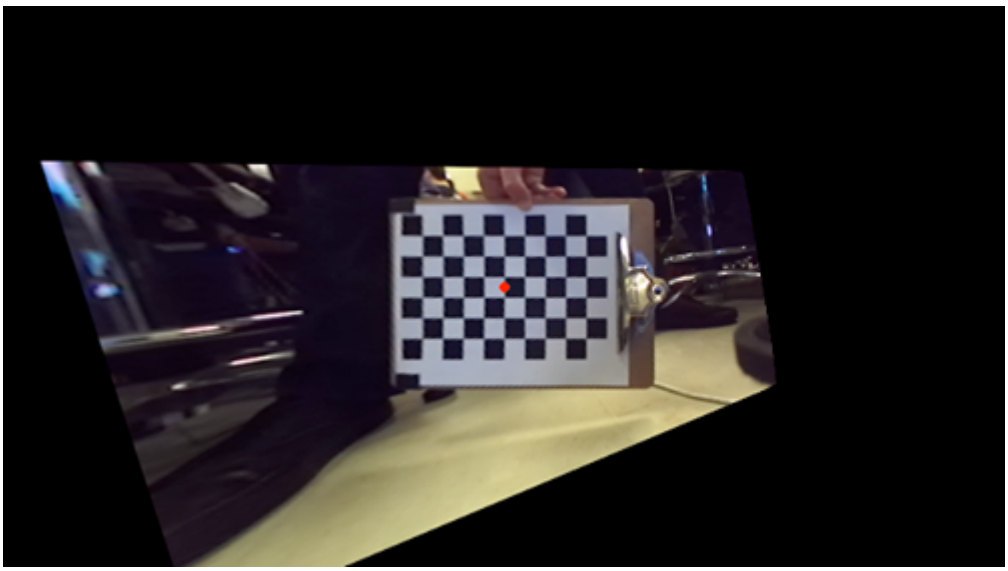
We construct a new matrix H that inverts the rotation and translation transformations done to the checkerboard: [3]

$$HC[R|\mathbf{t}]\mathbf{x} = C[I|\mathbf{0}]\mathbf{x}$$

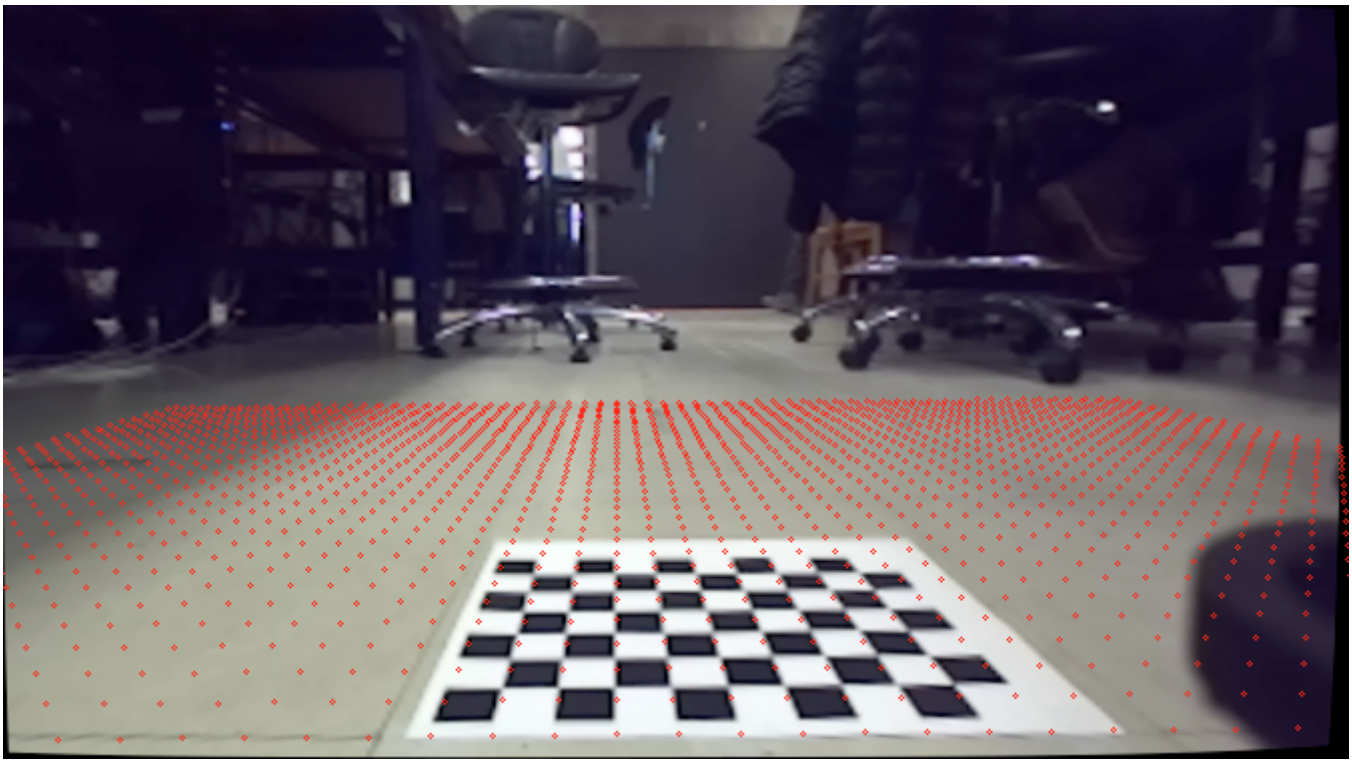
$H = H_R H_T$ is the product of matrices that invert the translation and rotations respectively:

$$
\begin{aligned}
H_R &= CR^{-1}C^{-1} \\
\mathbf{q} &= R^{-1}\mathbf{t} \\
H_T &= \left[\begin{array}{cc|c} 1 & 0 & \\ 0 & 1 & -\frac{C\mathbf{q}}{q_z} \\ 0 & 0 & \end{array}\right]
\end{aligned}
$$

This transformation matrix puts the origin at the top left corner of the chess board. To center it at the very center of the chess board we subtracted half of the width of the chessboard from the translation components of H_T. We also scaled the image to centimeters by setting the scale factor (the bottom right hand coordinte) of H_T to be $1/q_z$ rather than 1. The following image for demonstration purposes centers the chessboard in the image (not at the origin) and scales to fit in the image (not to centimeters).



We used the following image to define the rotation and translation of the floor. The resulting transformation matrix is then stored as a constant in our code. After performing this transformation we add the distance from the center of this chessboard to the rear axel to the forward direction. The mage is overlayed with a square grid of width 2.45 cm (the width of a single chess square) produced by our transformation matrix.



The camera matrix could successfully determine distances on the floor within a centimeter in a range of at least 2 meters in front of it.

These transformations are included in `CoordinateTransformations.py` in the Appendix.

[Trevor]

Pure Pursuit

The goal of Pure Pursuit is to find a goal point in real world coordinates at some set “lookahead” distance and use the Ackermann Steering model of the car to drive to that point at constant steering angle and velocity. The goal point is defined as the intersection between the line and a circle with radius equal to the lookahead distance centered at the rear axle.

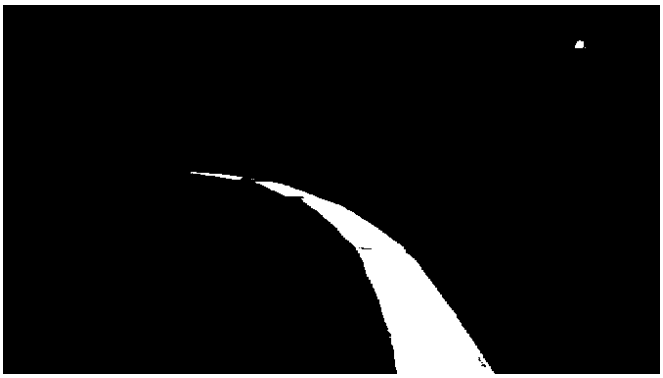
To achieve this, we first apply the same thresholding filter we used in cone detection to mask the tape from the background. As seen below, the same thresholds give good results (and even detect the cone on the table in the background) and return on the order of 10,000 points in the mask.

The robot needs to find a point on the tape that is at the lookahead distance from the rear axle. To find the goal point in the image, we first convert coordinates in the mask into real world coordinates using the camera transformation matrix described above. However applying the matrix transformation is expensive and it creates a huge bottleneck in our system when its run on all 10,000 points. To reduce the number of points that must be processed with the transformation matrix, we find the bounding contour of the mask using

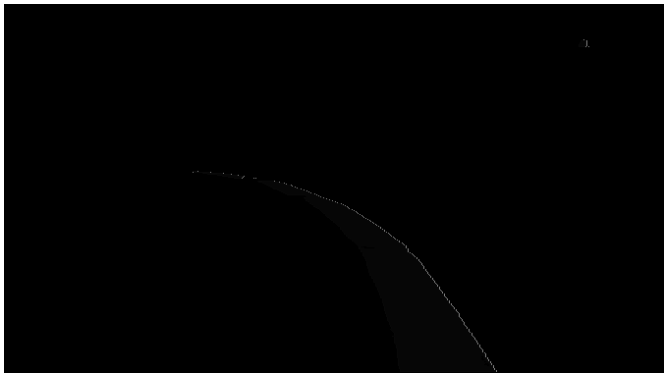
OpenCV's `findContours` method. This returns points along one side of the tape mask as shown below. To further reduce processing, our implementation only applies the camera transformation to 50 randomly sampled points along the contour. This gives 50 points along the tape and their respective transverse and longitudinal distances from the center of the rear axle.

The goal point is determined by finding the point among the randomly sampled points whose Euclidean norm is closest to the lookahead distance set in the Pure Pursuit algorithm. This is found by an iterative search. For the sake of demonstration, we apply the inverse of the transformation matrix to the goal point to map the goal point back to pixel coordinates and plot the goal point's pixel coordinates on the original image with a blue circle. As shown below, the algorithm chooses a point on the tape at a set distance from the rear axle of the car.

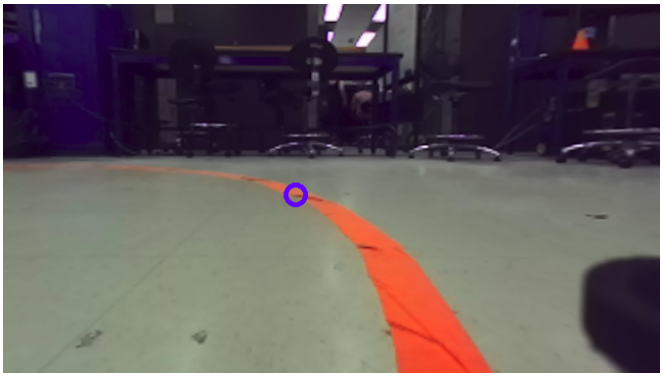
Original image



Threshold mask

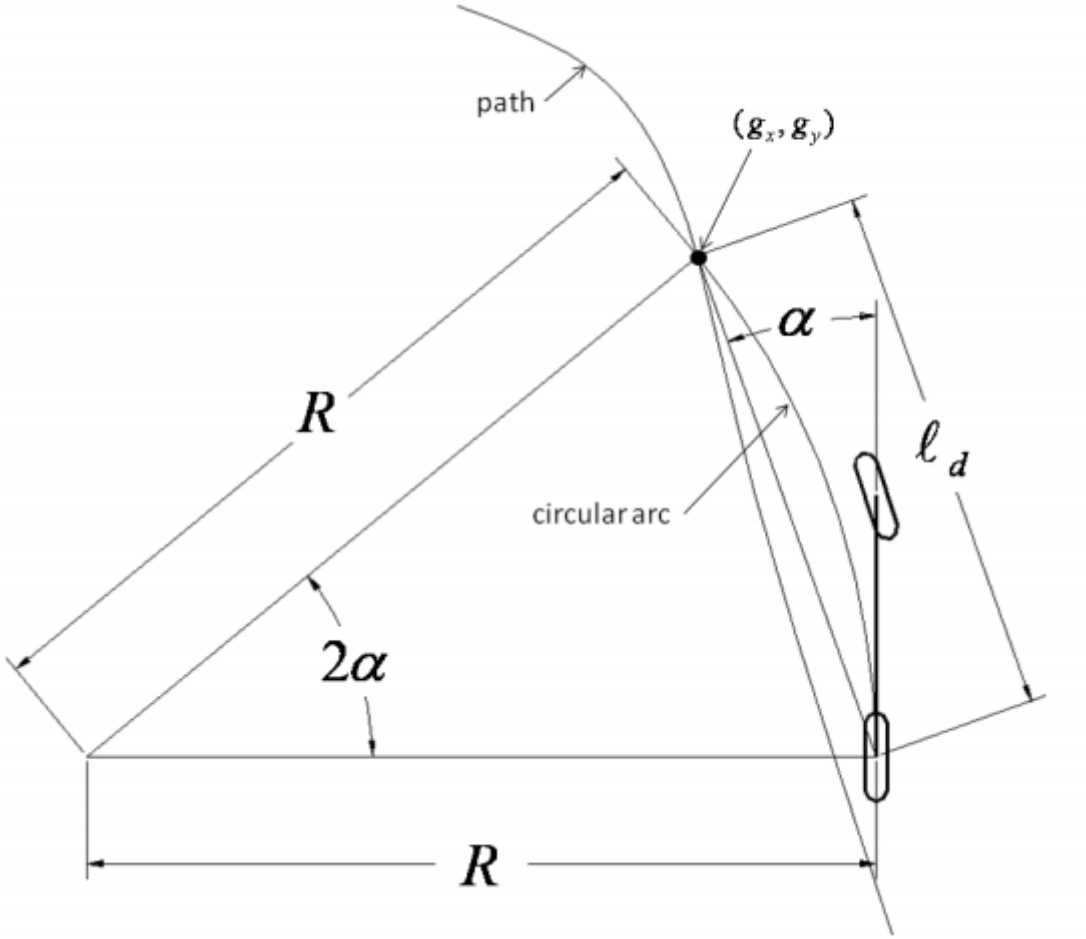


Contour of mask



Chosen goal

So far, our algorithm has produced a goal point for the car to drive towards given as a forward and horizontal distance in centimeters from the rear axle. With this goal point (shown as (g_x, g_y) below), we set the steering angle of the car by the simplified bicycle Ackermann model. [4]



The curvature of the car is

$$\kappa = \frac{2\sin(\alpha)}{l_d}$$

Where we can compute

$$l_d = \sqrt{g_x^2 + g_y^2} \qquad sin(\alpha(t)) = \frac{l_d}{g_x}$$

With a wheelbase of $L = 32.5cm$, we can find steering angle δ to be

$$\delta(t) = \arctan(\kappa L)$$

We command steering angle δ and drive at a constant velocity of our choosing. The lookahead distance and the speed of the car should be adjusted together- the faster the car is going, the further the goal point should be down the path. We thus set the quotient of the lookahead distance and the velocity of the car with a constant `reactionTime`. Through experimentation, we found that setting the car's velocity to 2.5 meters per

second and its `reactionTime` to 0.4 seconds gave the best stability but also highest speed while driving around the circle track. The Pure Pursuit control code is provided in the Appendix in `PurePursuit.py`.

[Kevin, Trevor]

Line Follower Results

Quantitative Line-following Performance
In testing, we tried to assess the following:

1. How many times can the car drive around the circle without colliding with the inner obstacle or desks around the circle?
2. What's the fastest speed it can go without collisions?

With open loop control, the car could not complete a single lap without colliding with obstacles around the circle. When we introduced feedback control with the setpoint controller and Pure Pursuit controller, the system was much more stable. As long as the set speed was not too high, both systems could drive around the circle indefinitely.

Method	# of Circles Completed	Maximum Stable Speed
Open-Loop	0.25	0.4 m/s
Setpoint	5 [turned off]	0.46 m/s
Pure Pursuit	5 [turned off]	2.23 m/s

The pure pursuit controller went fastest at 2.23 m/s as opposed to the setpoint controller which only went 0.466 m/s. The pure pursuit controller is able to go much faster because it looks much farther ahead of the curve than the setpoint controller.

We tried to push the Pure Pursuit model to go even faster. At 3 m/s, our robot managed to get around the circle once before spinning out of control. We believe that at such high speeds going around a circle with such a small radius, the centripetal force is causing the wheels to slip out. The Ackermann model is very simple and does not account for slipping wheels, so we believe it is breaking down at these high speeds. Changing the Ackermann model to account for slip could probably get our car to go faster.

[Katy, Kevin, Trevor]

Teamwork

Both portions of the lab allowed for division of responsibilities and practice in collaboration throughout the technical work. In lab 4A, the function of the controllers was essentially the same so the division first came from splitting off to attempt different cone detection implementations. This part of the lab definitely became an important practice for the team because it would require ability of teams to evaluate which cone detector worked best without bias of having worked on a different implementation. It helped that going into the process we all identified that this consolidation process would be required, and even expected a certain one to have the best performance. The choice was also easier because over the past few labs the team has been able to come together to work with everyone wanting the robot to operate with the best possible functionality, and using a technical filter to avoid any personal bias.

Part 4B of the lab had three defined methods of line-following to implement and compare to each other, so the team was able to divide and conquer each the open-loop, setpoint, and pure pursuit controllers. One person each worked on the first two because they were simpler to execute, and everyone else collaborated to execute the pure pursuit method. There was more collaboration during the debugging process and in running the robot through the required experiments.
[Caroline]

Conclusion

Overall, each aspect of the lab was successful. For part 4A, in terms of ability to complete the task, we were able to successfully maintain a set distance from the cone, and park in front of it. The robot was generally responsive, and could follow the cone both forwards and backwards. Even from a relatively large distance > 8ft, and at an angle, the car was able to locate and park in front of the cone. In part 4B, all line following methods were implemented successfully so as to evaluate the relative performances. It was determined through qualitative and quantitative experimentation that the Pure Pursuit implementation was the most responsive and controllable method for line following, even at high speeds.

An additional important take-away from this lab project was the camera calibration process, which can be utilized again in the future. Though it took a lot of time, we have a very accurate distance estimation capability.

The team website has continued to be updated. The most recent updates were to upload project images to the gallery pages, continue to update asthetics, and to add information about the class and projects.

[Josh, Caroline]

Appendix

Full code available upon request.

Pure Pursuit Code Snippets

```
#CameraSettings.py

#Stores constants for camera transformation matrix and car dimensions
#found in offline processing

import numpy as np

TRANSFORMATION_MATRIX = np.array(
  [[  9.97483827e-01,  -2.81893211e-01,  -2.64944973e+02],
   [  1.95516554e-02,   2.16788505e+00,  -6.71284167e+02],
   [  5.15496517e-04,   5.68335936e-02,  -8.26420446e+00]])

DISTANCE_FROM_BACK_WHEEL= 53.3+9.6 #cm
DISTANCE_FROM_CENTER = -1.3 #cm

```

```
#CoordinateTransformations.py

#Applies transformations to pixel values to obtain world coordinates
#given camera transformation matrix

#...

class CoordinateTransformations:

    def __init__(self, transformationMatrix, distanceToBackWheel, distanceTo
        self.transformationMatrix = transformationMatrix
        self.inverseTransformationMatrix = np.linalg.inv(transformationMatri
        # in CM
        self.distanceToBackWheel = distanceToBackWheel
        self.distanceToCenter = distanceToCenter

    # (0,0) is the back wheel
    # measurement is in CM
    def transformPixelsToWorld(self, pixelCoordinates):
        affinePixelCoordinates = [pixelCoordinates[0], pixelCoordinates[1],
        affineWorldCoordinates = np.matmul(self.transformationMatrix, affine

        worldCoordinates = [affineWorldCoordinates[0]/affineWorldCoordinates
        worldCoordinates[0] = worldCoordinates[0] + self.distanceToCenter
        worldCoordinates[1] = -worldCoordinates[1] + self.distanceToBackWhee

        # returns measurement in cm
        return worldCoordinates

    # ...
```

```
# CircleThreshold.py

# Thresholds image and finds goal point in contour of threshold mask
# PurePursuit.py makes call to findGoalPoint()

from CoordinateTransformations import CoordinateTransformations
from CameraSettings import *
# ...

class CircleThreshold:
    RANDOM_SAMPLE_SIZE = 50

    def __init__(self, lookAheadDistance):
        self.lookAheadDistance = lookAheadDistance
        #lookahead distance in cm
        self.mask = None

        self.CoordinateTransformations = CoordinateTransformations(\
            TRANSFORMATION_MATRIX, DISTANCE_FROM_BACK_WHEEL, \
            DISTANCE_FROM_CENTER)

    #applies cv2.inrange filter to image, stores mask in self.mask
    def threshold(self, inputImage):
        # ...

    #applies transformation to points on contour and saves real
    #world coordinates to self.circle
    def transform(self):
        #...

    def findBestContour(self):
        _, contours, _ = cv2.findContours(\
                self.mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
        self.contourPoints = [point for contour in contours\
                                for point in contour]

    def findGoalPoint(self, inputImage):
        self.threshold(inputImage)
        self.findBestContour()
        self.transform()

        closest = None
        for (x, y) in self.circle:
            if closest:
                if abs(np.linalg.norm((x, y))-self.lookAheadDistance) < \
                    abs(np.linalg.norm(closest)-self.lookAheadDistance):
                    closest = (x, y)
            else:
                closest = (x, y)

        return closest
```

```
#!/usr/bin/env python

# PurePursuit.py

# Calls on CircleThreshold.py to find goal point and then uses
# Ackerman model of RACECAR to steer at constant angle to goal.

from CircleThreshold import CircleThreshold
# ...

class PurePursuit:
    carLength = 32.5 # cm
    velocity = 2.5 # m/s
    reactionTime = 0.4  # sec
    lookAheadDistance = velocity * 100 * reactionTime

    def __init__(self):
        self.bridge = CvBridge()
        self.sub_image = rospy.Subscriber("/zed/rgb/image_rect_color",\
                Image, self.PureControl, queue_size=1)

        self.publisher = rospy.Publisher(\
                "/vesc/high_level/ackermann_cmd_mux/input/nav_0",\
                AckermannDriveStamped,\
                queue_size = 1)

        self.pub_image = rospy.Publisher("/echo_image",\
                Image, queue_size=1)

    # Handler for image messages from camera
    def PureControl(self, image_msg):
        image_cv = self.bridge.imgmsg_to_cv2(image_msg)

        ct = CircleThreshold(self.lookAheadDistance)
        goalPointWorld = ct.findGoalPoint(image_cv)

        steeringAngle = self.ackermannAngle(goalPointWorld)
        msg = AckermannDriveStamped()
        if steeringAngle:
            rospy.loginfo('steeringAngle = %f', steeringAngle)
            msg.drive.speed = self.velocity
            msg.drive.steering_angle = steeringAngle
        else:
            rospy.loginfo("no line!")

        self.publisher.publish(msg)

    # Return steering angle to goalPoint given by simplified
    # Ackermann model of car.
    def ackermannAngle(self, goalPointWorld):
        # ...
```

```
52
53  # ...
```

Sources

1. http://aishack.in/tutorials/sift-scale-invariant-feature-transform-introduction/ (http://aishack.in/tutorials/sift-scale-invariant-feature-transform-introduction/) ↩

2. http://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html (http://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html) ↩

3. http://stackoverflow.com/questions/23275877/opencv-get-perspective-matrix-from-translation-rotation (http://stackoverflow.com/questions/23275877/opencv-get-perspective-matrix-from-translation-rotation) ↩

4. https://www.ri.cmu.edu/pub_files/2009/2/Automatic_Steering_Methods_for_Autonomous_Automobile_Path_Tracking.pdf (https://www.ri.cmu.edu/pub_files/2009/2/Automatic_Steering_Methods_for_Autonomous_Automobile_Path_Tracking.pdf) ↩