Racecar Project

1. Introduction

   The racecar project is trying to give out and verify a **simple** solution for autonomous racing, where racecars are to navigate inside a track marked by red and blue cones. The project is really simple since it requires simple sensor (a mono camera is enough), utilizes simple algorithms/tools in perception, path planning and control, and has simple code structure for readers to understand and adapt for their own use. The following paragraphs will discuss the autonomous navigation task and solution in detail

2. Task setup

   Here is an image showing the track

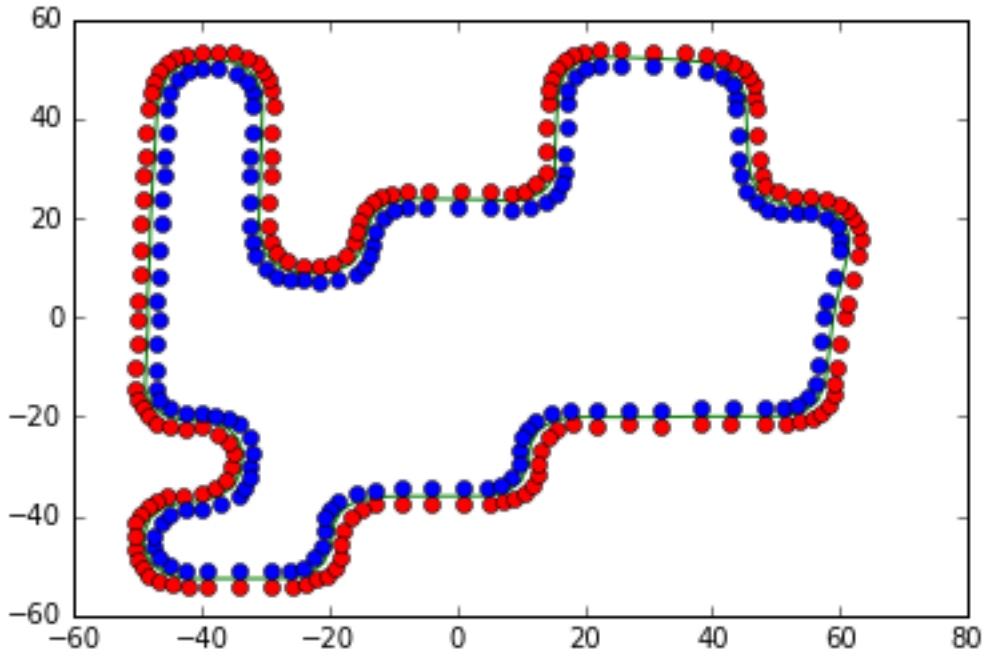

Figure 1: top view of a track marked by red and blue cones

As is shown in Figure 1, the track is marked out by red and blue cones, red cones are specifying the left track, while blue cones specifying the right. Straight lanes, S-curves and U-curves are included. Distance between adjacent cones with same color won't exceed 5 meters. The narrowest part of the track has a distance larger than 3 meters.

3. Project setup

Five packages are included in this project. Package "eufs_description" and "eufs_gazebo" include definitions of racecar and simulation environments. Package "sim_simple_controller" controls the racecar in gazebo by manipulating incoming speed command. The package "auto_pilot_param" is the main package

dealing with perception, path planning and generating speed command. The script package contains a server for perception (will discuss later in section 4), and other scripts for installation.

Here is a diagram showing the project structure

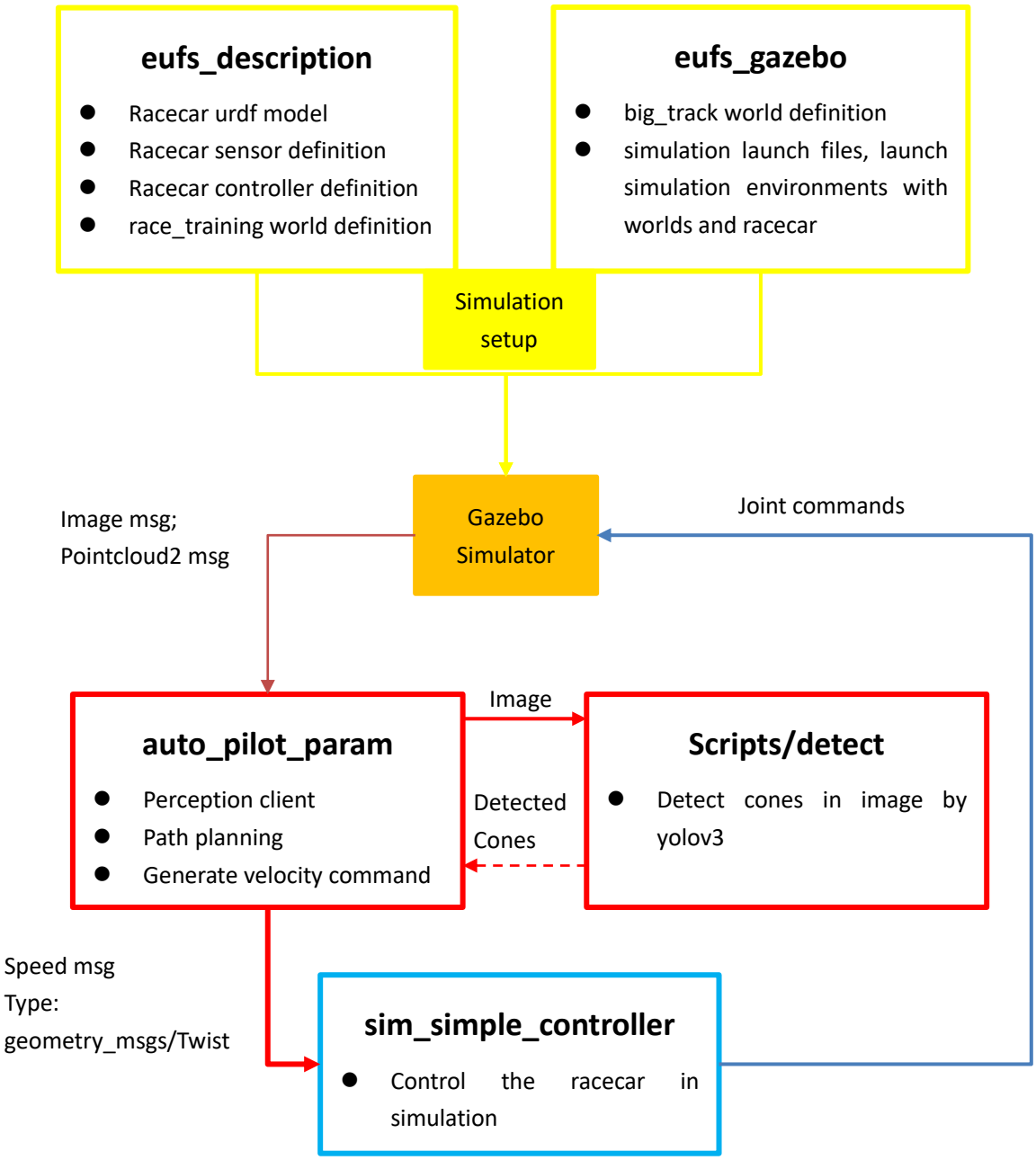

Figure 2: project package structure diagram

Package "eufs_description", "eufs_gazebo" and "sim_simple_controller" are all adapted from [eufs_sim] and are not the main focus of this report. Instead, I will focus on the "auto_pilot_param" package and "scripts/detect" package, where perception, path planning and speed command generation take place.

4. Coordinate system definition

Before we can proceed to discuss about perception, path planning and control, we need to define several essential coordinate systems in the first place.

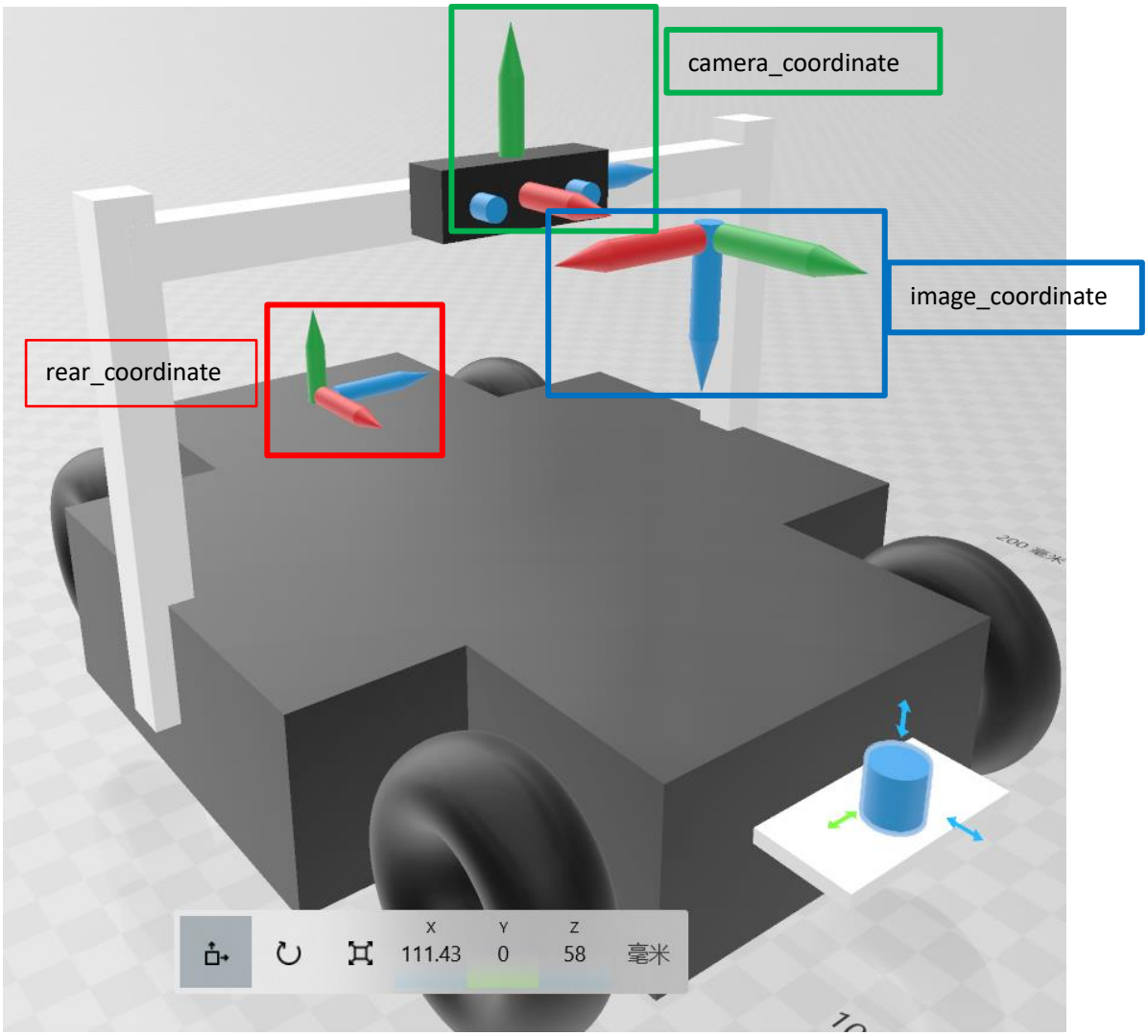

Figure 3: coordinate system definitions.

Red arrows in the figure are showing x-direction, blue arrows are showing y-direction and green arrows are showing z-direction. The coordinate bounded by a red box is rear coordinate, whose origin is at the middle of the line connecting the rear tires. The x-direction of the rear coordinate is pointing at the direction of navigation. The coordinate bounded by a green box is the camera coordinate, whose origin should be at focal point. The coordinate bounded by the blue box is the image coordinate, whose origin is at the left-upper corner of the image. Note that when we are taking images, the z-axis of the image coordinate vanish since images are 2D.

5. Naïve Perception Solution

As has been stated plenty of times above, the track the racecar navigates in is marked by red and blue cones. The main perception task is therefore getting 3D position of red/blue cones in terms of the rear coordinate (the control system is based on the rear coordinate, will discuss later).

Using pointcloud information generated by 3D-lidars like velodyne may be the easiest way to get 3D cone positions. However, the 16-line lidar I have in simulation only gives pointcloud within 5-6 meters, making the idea of using lidar data infeasible. Therefore, my solution is to get the 3D cone positions by only using camera image.

Step 1: Get 2D cone position inside the image

To detect cones in image, I trained a model based on yolov3, and adapted the detect.py to give the middle point of the cone bounding box as well as the color of the cone. The implemention can be find in scripts/detect/cone_detect.py.

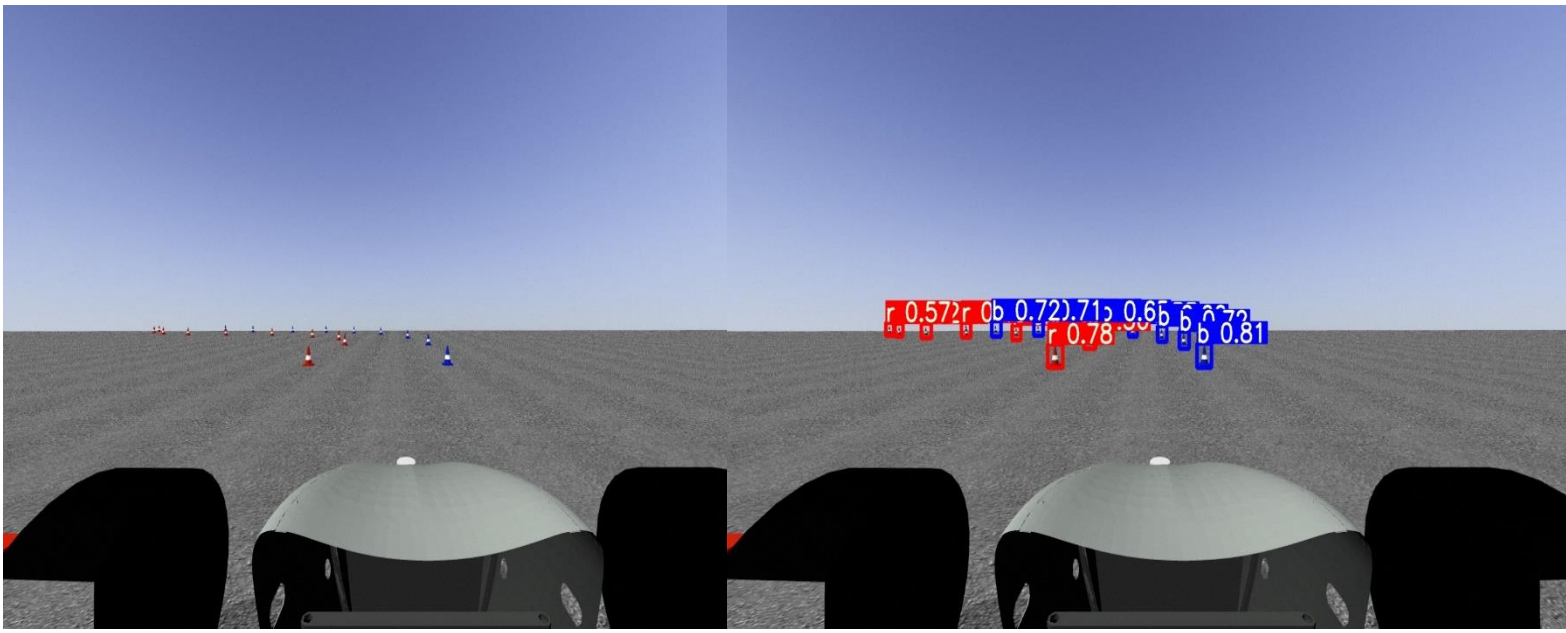

Figure 4: demo of cone detection by yolov3, left is the input image, right is the detection result. The center of each bounding box will be returned by cone_detect.py.

Note that the scenario in simulation is simple, and the yolov3 model I trained with a dataset of 800 images within 100 iterations. In real world, it may be much harder to train the model.

Step 2: Get 3D cone position based on 2D cone center point

According to the basic pinhole camera model (相机模型成像原理 if you read Chinese),

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{Z} \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

where X,Y,Z are x,y,z coordinate value in terms of the image coordinate defined in section 4; f_x, f_y are focal length of the camera, and c_x, c_y are the coordinate number of the focal point in the x-y plane of the image coordinate; u,v are the projection coordinate of the 3D point defined by X,Y,Z, it is impossible to recover the value of $\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$ when we only know $\begin{bmatrix} u \\ v \end{bmatrix}$.

The trick here is that the height of cone is fixed and known. Assume the height of the camera with respect to the ground won't change too much, then we have a known Y value, and can solve the system above and recover $\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$.

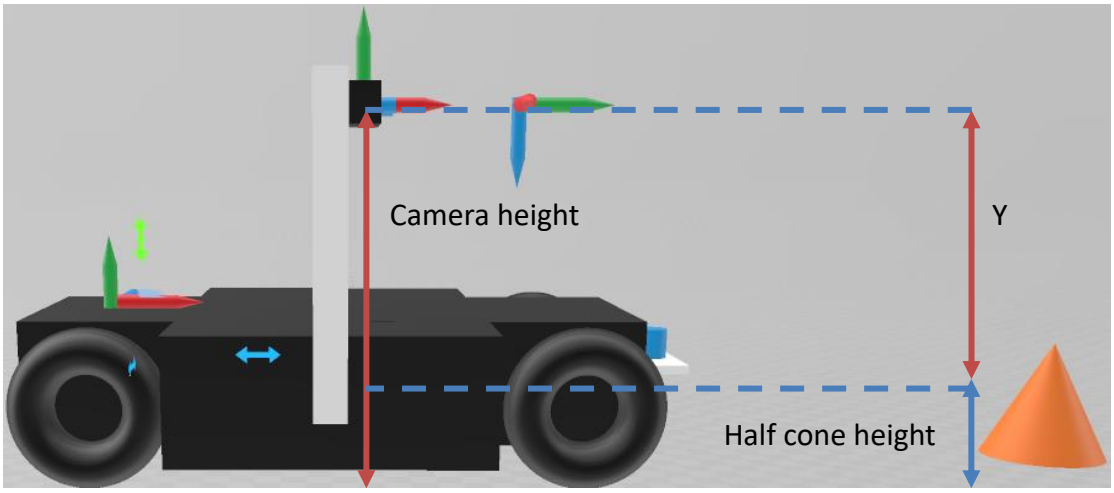

Figure 5: illustration of Y derivation. Y = camera_height – cone_height / 2

Note: this may be the most troublesome part of the project. The detected 3D cone position may shake from time to time. If your sensor permits, use other sensors to measure cone position and fuse the result with this method. (E.g. use 3D lidar to detect cones and then use the camera model above to project the pointcloud onto camera x-y plane to get cone color).

After getting the $\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$ value in terms of image coordinate, we can apply

coordinate transformation to get the $\begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix}$ values in rear coordinate. The implementation of getting 3D cone position from 2D image can be found in pixel2car_param.py.

Step 3: Data transmission

As described above, the cone detection is done by using yolov3 model with custom parameters. However, Yolov3 is designed for python3, while the simulation environment is depending on ROS-Kinetic/Melodic, whose python version is python 2.7. Therefore, I cannot use yolov3 directly within a ROS node for ROS-Kinetic/Melodic.

To solve this problem, one possible solution is to use ROS2 together with ROS. However, this solution may make the project structure complex, and is unfriendly to those who are not familiar with ROS2.

My solution here is to use python socket interface to do the data transmission, as shown in the diagram below:

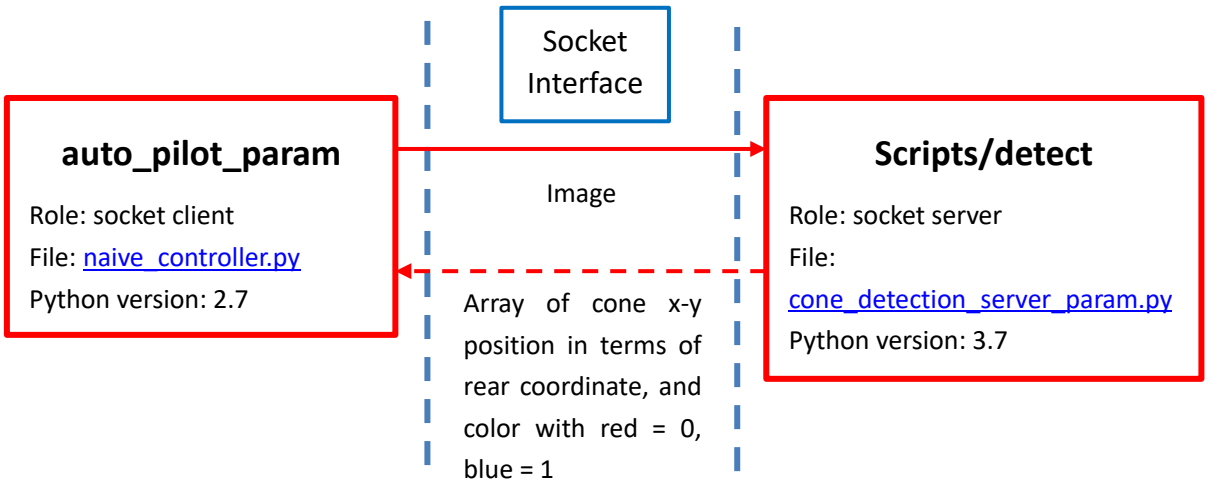

Figure 6: illustration of perception data transmission. A socket client is created in auto_pilot_param package, sending the images it receive to the socket server in scripts/detect. The server will detect cones and do the transformation from 2D to 3D, and then send the cone position and color back to the client.

Finally, we classify cones with color (red/blue), getting two arrays of red_cone/blue_cone x-y coordinate numbers in terms of rear coordinate.

6. Naïve Path Planning Solution

After the perception step, have two arrays of red_cone/blue_cone x-y coordinate numbers in terms of rear coordinate. The task now is to form a path for the racecar to follow based on these cones.

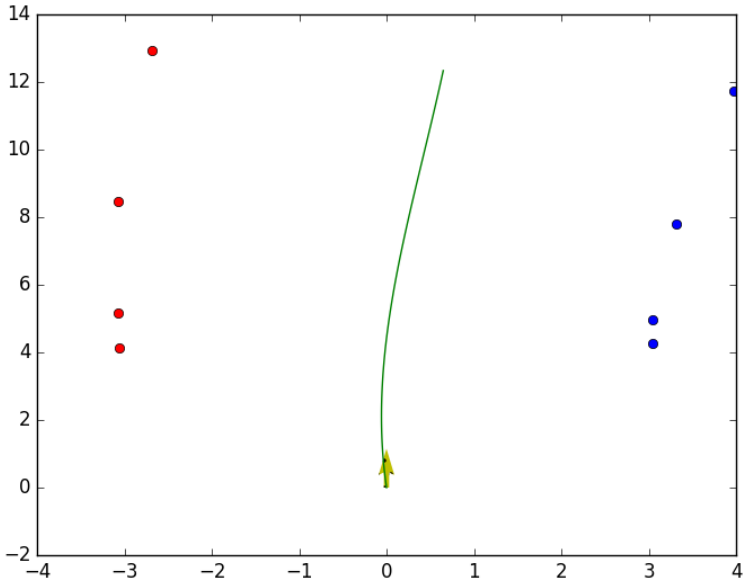

Figure 7: demo of path planning based on red/blue cones in car rear coordinate. The detected red/blue cones are shown as red/blue dots, the yellow arrow is the unit vector representing x direction of car rear coordinate. The green curve in the middle is the path for the car to follow.

Basic idea:
As described in section 2 Task setup, the red cones are marking the left lane, while the blue cones are marking the right lane. To form a feasible path, we can simply take the average of the left and right lane point. The pseudo code is as follows:

```
Function form_path_basic(red_cones_2d, blue_cones_2d):
    path = {}
    If len(red_cones_2d) ≥ len(blue_cones_2d) then
        for all point in blue_cones_2d do
            nearest_neighbor = find_nearest_neighbor(point, red_cones_2d)
            path = path ∪ {(points + nearest_neighbor) / 2}
    Else
        for all point in blue_cones_2d do
            nearest_neighbor = find_nearest_neighbor(point, red_cones_2d)
            path = path ∪ {(points + nearest_neighbor) / 2}
    path = link_path_points(path)

    return path
```

The task of finding nearest neighbor is done by using KDTree datastructure. And the implementation of link_path_points can be found in generate_path2.py.

The idea above is really simple, yet it works in most cases.

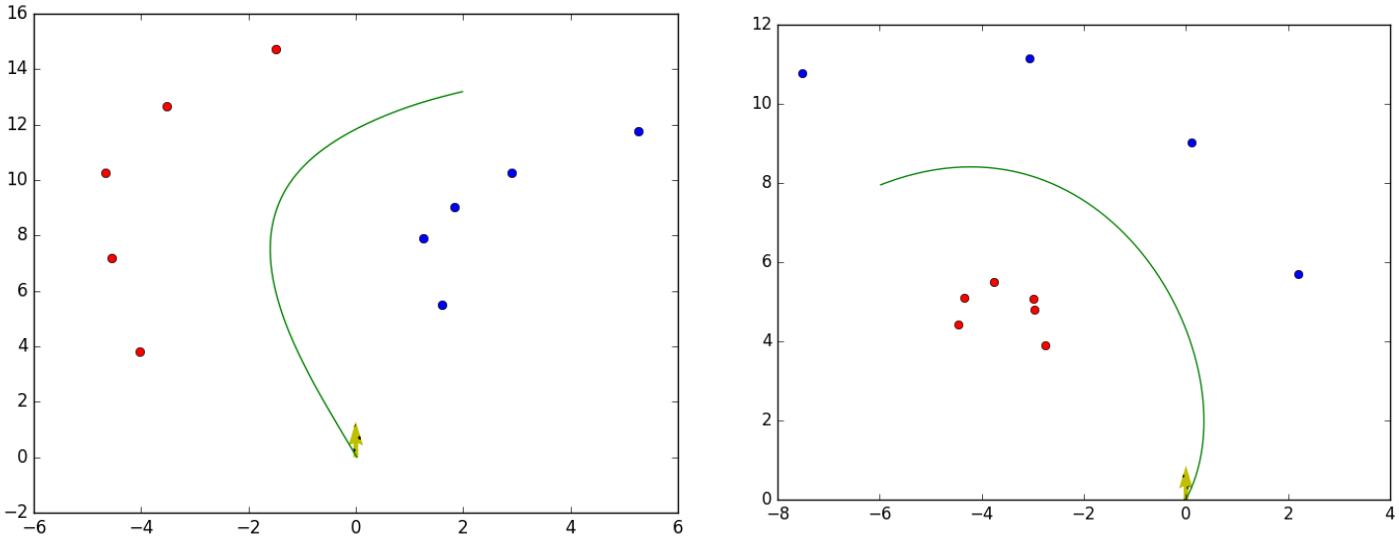

Figure 8: Demo of the simple path formation idea.

Corner case 1: few red/blue cones

Some times, cones of a color is detected multiple times, while cones of the other color is detected only once or even vanishes.

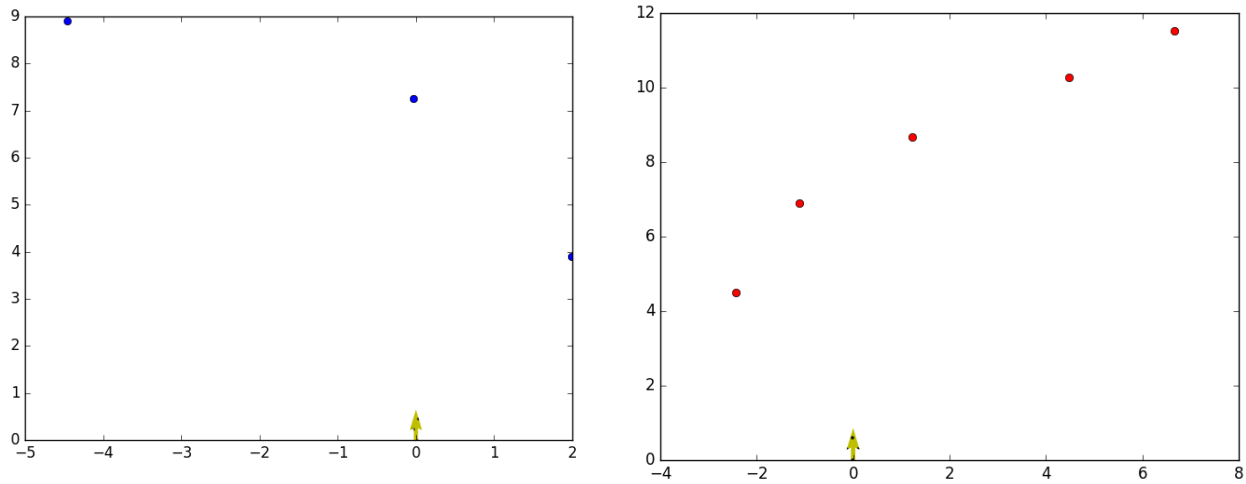

Figure 9: left figure shows the case where too few red cones is detected, right is the case for blue cones.

Since the red cones are marking the left lane and blue cones are marking the right lane, when we have too few red cones, the racecar will turn left; and when we have too few blue cones, the racecar will turn right.

Corner case: Too much red/blue cones

Sometimes, cones from unrelated region may also be detected. If these cones are included for path planning, the planned path may be troublesome. Here is an example:

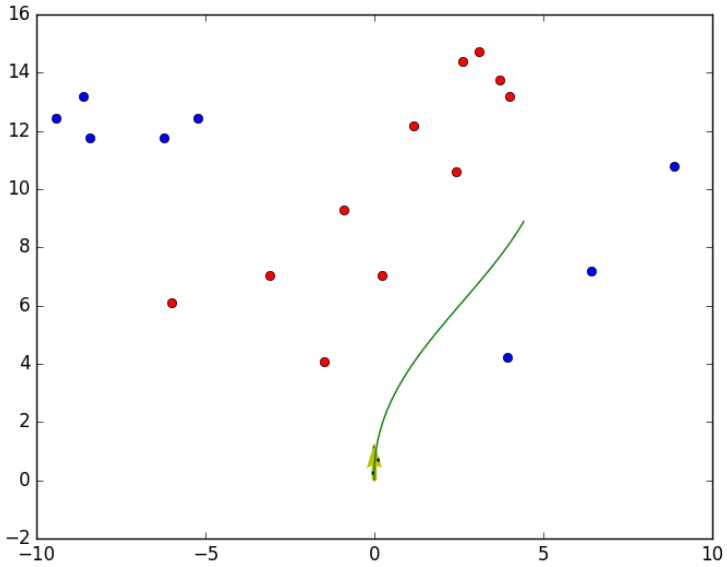

Figure 10: on the left corner of the figure, 5 extra blue points are detected. If these points are included in path planning using the basic idea described above, the formed path will go through the red cones.

To solve this problem, I introduced a "classification check" to the formed path. In theory, the path we planned should be able to classify the red and blue cones into two region. If misclassification happens, we abandon cones too far away from the origin and form the path based on new points again. The classification is done based on the property of the curve

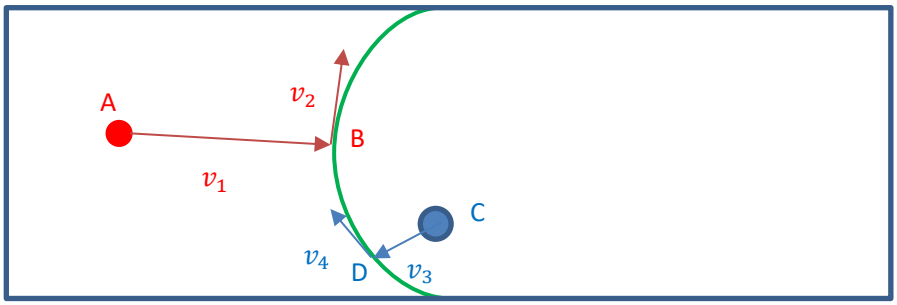

Figure 11: illustration of how curve classification check works. Suppose the green curve in the middle splits the region into 2 parts. Point A is to the "left" of the curve, with B being its closest point on the curve and v_2 being the tangent vector of the curve at B. Point C is to the "right" of the curve, with D being its closest point on the curve and v_4 being the tangent vector of the curve at D. A,B,C,D are 2D points and v_1, v_2, v_3, v_4 are all 2D vectors. Suppose we add a zero

Z-dimension to v_1, v_2, v_3, v_4,for example, ${v_1}' = \begin{bmatrix} v_{1x} \\ v_{1y} \\ 0 \end{bmatrix}$, and then do cross product: $C_1 = v_1' \times v_2'; C_2 = v_3' \times {v_4}'$, then the Z dimension of C_1 should be positive, while the Z dimension of C_2 should be negative. A,C are arbitrarily chosen point, and this property can be applied to any point in the field.

The implementation of "classification check" can be found in naive_controller.py, line 228 – 255.

Here is a diagram of the state machine for path planning and speed control:

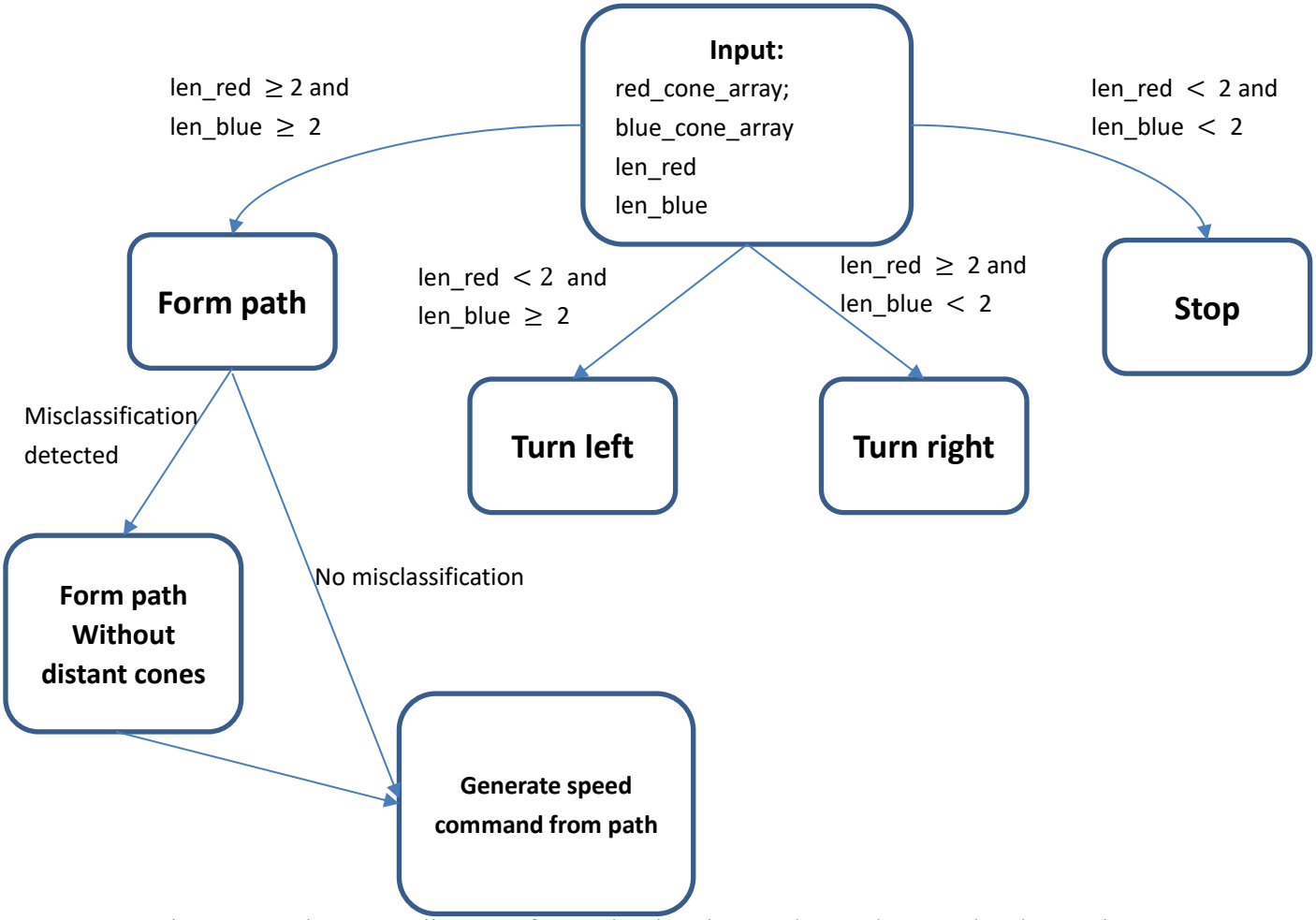

Figure 12: the state diagram for path planning and speed control. The project forms path only when enough cones are detected, and it generate speed command from path only when a path is present.

7. Generate speed command
   Two steps are required for generating a speed command.

   Step 1: smooth the input path
   As described above, the input path is generated by averaging the red and blue cone coordinates. The number of points included is too few. To solve this issue, I used UnivariateSpline to generate a smooth path. Note that it is impossible to do spline on a curve directly. Instead, we need to calculate the cumulative sum s along the path and spline based on "s(x)" and "s(y)". See "how to interpolate a 2D curve in python" for detail.

   Step 2: generate speed command
   After having a smoothed path, I used the pure-pursuit control model to generate speed command from input path. The implementation for path smoothing and applying pure-pursuit control model can be found in pure_pursuit_wrapper.py.

8. Tutorial for real-world use

   As shown in section 3, only "auto_pilot_param" package and "scripts" package in this project are for perception, path planning and control. Therefore, to apply this project in real world, you only need to include these two packages. Here are a few steps for adapting the project for real world use:

   Step 1: change parameters to fit your system
   All parameters used in the racecar project are included in a single file called /auto_pilot_param/params/racecar_params. The file is densly commented and changing it should be easy

   Step 2: train your own cone detection parameters
   The cone detection parameters used in this project cannot be applied to real world cases. Please train your own version of parameters and put it in /scripts/detect folder. In addition, remember to change the parameter name inside the parameter file in step 1.

   Step 3: prepare your own racecar base controller
   The racecar project advertises speed command in geometry_msgs/Twist type, including the linear speed and steer angle for vehicle control. Please check the behavior in sim_simple_controller package and make sure your base controller follows the same convention as in sim_simple_controller. (You can run the rqt_robot_control.launch) to see effect of input commands.

   In addition, please make sure your racecar is equipped with emergency stop to ensure safety.

   Shiji Liu 1/21/2021