ARL PLANNING TASK

Doha Mohamed

Senior 1 CSE

2200911

Contents

Overview

This path planner generates a drivable path for an autonomous car navigating between cones. The cones are detected and classified by color:

- **Blue cones** → left side of the track
- **Yellow cones** → right side of the track

The goal of the planner is to compute a sequence of waypoints (x, y) that define a smooth and safe **centerline path** between these cones, even when only one or no sides are visible.

Inputs

- car_pose: the current position and orientation of the car (x, y, yaw)
- cones: a list of Cone objects with:
  - x, y: cone coordinates in the world frame
  - color: 0 for yellow (right side), 1 for blue (left side)

Outputs

- Path2D: a list of (x, y) coordinates representing the planned path in world coordinates. The path is continuous, smooth, and extends 5–10 meters forward with step size ≤ 0.5 m.

Algorithm

Cone Separation & Sorting by Distance

– All cones are divided into left (blue) and right (yellow) sets based on their color labels.
– Each cone set is sorted by Euclidean distance from the car. To ensure that closer cones are processed first

```
#cones: yellow (right),blue (left)
yellow_cones = [c for c in self.cones if c.color == 0]
blue_cones = [c for c in self.cones if c.color == 1]

yellow_cones.sort(key=lambda c: self._dist(c))
blue_cones.sort(key=lambda c: self._dist(c))
```

Centerline Estimation (Midpoint)

This depend on which cones are visible.

Case 1: Both sides visible

– For each left-right pair, compute the midpoint between the blue and yellow cones:

```
# 1: both sides visible
if blue_cones and yellow_cones:
    num_pairs = min(len(blue_cones), len(yellow_cones))
    for i in range(num_pairs):
        mx = (blue_cones[i].x + yellow_cones[i].x) / 2
        my = (blue_cones[i].y + yellow_cones[i].y) / 2
        midpoints.append((mx, my))
```

– I assume cones on each side have matching order along the track (closest-to-farthest).

Case 2: Only one side visible

If only one side of cones is detected:

– I assumes an approximate track width of 3.0 meters.
– It guesses the missing cone positions by offsetting the detected cones laterally by this width, using the car's yaw to determine left/right direction.
– The midpoint between the detected cone and its guessed opposite cone defines the centerline point.

```
#  2: one side visible
elif blue_cones or yellow_cones:
    track_width = 3.0
    cones = blue_cones if blue_cones else yellow_cones
    for c in cones:
        if blue_cones:
            dx = math.cos(self.car_pose.yaw - math.pi / 2)
            dy = math.sin(self.car_pose.yaw - math.pi / 2)
            yellow_guess = (c.x + track_width * dx, c.y + track_width * dy)
            mx = (c.x + yellow_guess[0]) / 2
            my = (c.y + yellow_guess[1]) / 2
        else:
        # Guess missing blue cone
            dx = math.cos(self.car_pose.yaw + math.pi / 2)
            dy = math.sin(self.car_pose.yaw + math.pi / 2)
            blue_guess = (c.x + track_width * dx, c.y + track_width * dy)
            mx = (c.x + blue_guess[0]) / 2
            my = (c.y + blue_guess[1]) / 2

        midpoints.append((mx, my))
```

Case 3: No cones visible

- If no cones are detected, the car drives straight ahead 3 meters in the direction of its current heading.

```
#  3: no cones visible
else:
    mx = self.car_pose.x + 3.0 * math.cos(self.car_pose.yaw)
    my = self.car_pose.y + 3.0 * math.sin(self.car_pose.yaw)
    midpoints.append((mx, my))
```

3)

Path Construction

- The car's current position is added as the **starting point** of the path.
- Consecutive midpoints are connected with **linear interpolation**, ensuring smooth transitions between them.
- Interpolation uses ~10 steps per segment . (t ranges 0..1 inclusive)

```
# Start path from car position
start = (self.car_pose.x, self.car_pose.y)

if midpoints:
    path.append(start)

    # Connect all midpoints smoothly
    for i in range(len(midpoints) - 1):
        x0, y0 = midpoints[i]
        x1, y1 = midpoints[i + 1]
        steps = 10
        for j in range(steps):
            t = j / (steps - 1)
            x = x0 + t * (x1 - x0)
            y = y0 + t * (y1 - y0)
            path.append((x, y))
```

Path Extension

– To ensure the path extends forward(the car always has a forward trajectory) beyond visible cones:

- The code computes the **heading** of the last segment (between last two midpoints or car yaw if only one midpoint).
- It adds 5 more waypoints spaced 0.5 meters apart in that direction.

```
# Extend forward
if len(midpoints) >= 2:
    x_last, y_last = midpoints[-1]
    x_prev, y_prev = midpoints[-2]
    heading = math.atan2(y_last - y_prev, x_last - x_prev)
else:
    heading = self.car_pose.yaw

step = 0.5
for k in range(5):
    x_ext = midpoints[-1][0] + step * k * math.cos(heading)
    y_ext = midpoints[-1][1] + step * k * math.sin(heading)
    path.append((x_ext, y_ext))
```

Distance Function

– _dist(cone) computes Euclidean distance between the car and each cone:

```
def _dist(self, cone: Cone) -> float:
    """Distance from car to cone."""
    return math.hypot(cone.x - self.car_pose.x, cone.y - self.car_pose.y)
```

Limitations & Improvements

Issue	Description	improvement
Fixed track width	Always assumes 3m	Estimate dynamically from detected cone pairs
Linear interpolation	Produces piecewise-linear path	Use cubic spline or polynomial fit for smoother curvature (Spline smoothing)
Pairing cones by distance	Can mismatch in curved sections	Pair based on projection along car heading
No curvature check	May produce sharp turns	Limit curvature by checking turning radius ;apply low-pass filter on heading changes

PART TWO

Approach

When three cones are detected on the same side (either all blue or all yellow):

1. **Estimate Local Track Direction**
    - Compute the **heading** between consecutive cones on that side using:

$$\text{heading} = \tan^{-1}\left(\frac{y_2 - y_1}{x_2 - x_1}\right)$$

    - This gives an approximate local direction of the track.
2. **Estimate Missing Opposite Cones**
    - Assume a **constant track width** (≈ 3 meters).
    - Project an imaginary cone on the **opposite side** by shifting perpendicular to the estimated heading.

For blue cones (left side):

$$x_{est} = x + w \cdot \cos(\text{heading} - \frac{\pi}{2})$$
$$y_{est} = y + w \cdot \sin(\text{heading} - \frac{\pi}{2})$$

For yellow cones (right side):

$$x_{est} = x + w \cdot \cos(\text{heading} + \frac{\pi}{2})$$
$$y_{est} = y + w \cdot \sin(\text{heading} + \frac{\pi}{2})$$

3. **Compute Midpoints**
    - For each real cone, calculate the midpoint between the actual cone and the estimated opposite cone.
    - These midpoints approximate the **centerline** of the track.
4. **Smooth Path Construction**
    - Interpolate between midpoints linearly to form a continuous path.
    - Extend the final segment slightly forward for a smoother exit.

```
elif len(blue_cones) >= 2 or len(yellow_cones) >= 2:
    track_width = 3.0
    side_cones = blue_cones if blue_cones else yellow_cones
    side_cones.sort(key=lambda c: self._dist(c))

    for i in range(len(side_cones) - 1):
        c1 = side_cones[i]
        c2 = side_cones[i + 1]

        # Estimate track direction between consecutive cones
        heading = math.atan2(c2.y - c1.y, c2.x - c1.x)

        # Offset to estimate opposite side
        if blue_cones:  # blue = left, offset to right
            dx = math.cos(heading - math.pi / 2)
            dy = math.sin(heading - math.pi / 2)
        else:  # yellow = right, offset to left
            dx = math.cos(heading + math.pi / 2)
            dy = math.sin(heading + math.pi / 2)

        # Midpoint between real and estimated cone
        mx = (c1.x + (c1.x + track_width * dx)) / 2
        my = (c1.y + (c1.y + track_width * dy)) / 2
        midpoints.append((mx, my))
```

Scenarios :

I put 3 scenarios to test it :

1) All cones are blue

```
"21": (
    [
        Cone(x=2.0, y=1.0, color=1),
        Cone(x=3.0, y=1.0, color=1),
        Cone(x=4.0, y=1.0, color=1),

    ],
    CarPose(x=0.0, y=0.0, yaw=1.4),
),
```

2) All cones are yellow

```
"22": (
    [
        Cone(x=2.0, y=1.0, color=0),
        Cone(x=3.0, y=1.0, color=0),
        Cone(x=4.0, y=1.0, color=0),

    ],
    CarPose(x=0.0, y=0.0, yaw=1.4),
),
```

3) Canes are mix between yellow and blue

```
"23": (
    [
    Cone(x=2.0, y=1.0, color=0),
    Cone(x=3.0, y=1.0, color=1),
    Cone(x=4.0, y=1.0, color=0),

    ],
    CarPose(x=0.0, y=0.0, yaw=1.4),
),
```

When I was doing this approach, I remember scenario number 5 which is two cones in same side this doesn't done a correct path in the first algorithm so I chang the condition hear to be >=2 instead of >=3

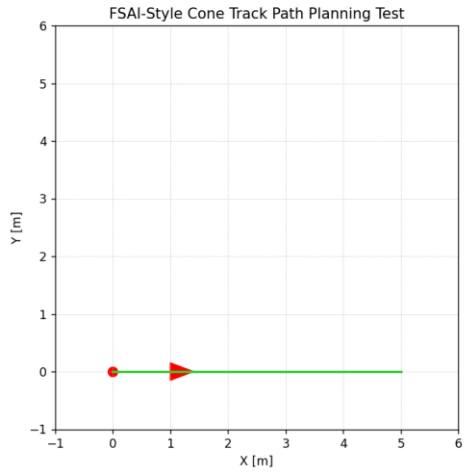
```
elif len(blue_cones) >= 2 or len(yellow_cones) >= 2:
```

Also I change the condition of the previous if to be correctly executed .

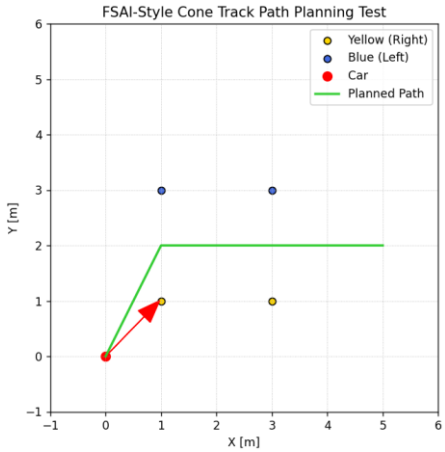
```
elif (blue_cones and len(blue_cones) < 2) or (yellow_cones and len(yellow_cones) < 2):
```

Some Running Scenarios :

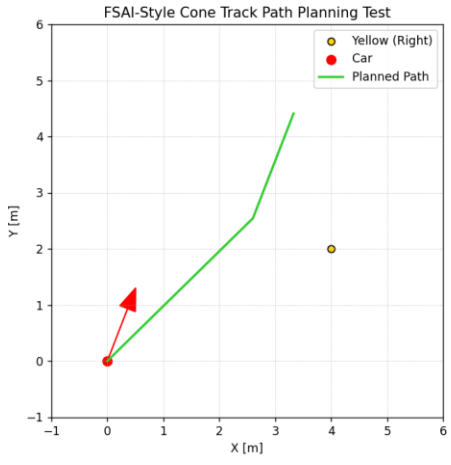
Scenario (1):



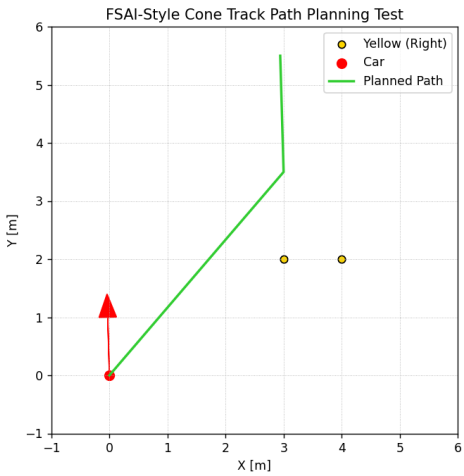
Scenario (3):



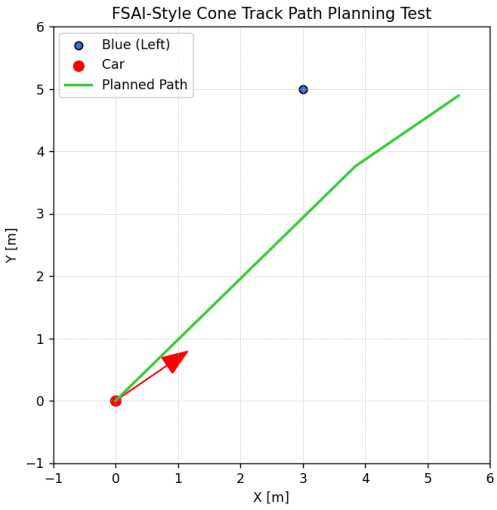
Scenario (4):



Scenario (5):



Scenario (17):



Scenario (22):