Nautilus Final Race Approach

Team Members: Stephanie Hughes, Prakash Baskaran, Zhicheng Xu (Stark)

Our group analyzed two methods for determining waypoints: pre-produced waypoint collection with smoothing and bi-directional RRT*.

1. Waypoints Smoothing

1.1 Generate original waypoints by using center wall-following

1.2 Replace the quasi-straight line on original waypoints with a straight line

Find the start and end waypoints for the straight line. We chose to straight the race line C1-M, F-G, H-I, J-K, K-A, and marked the index in the original waypoints for reference. In Fig.1 the index is followed by the letter. In the meanwhile, the raceline close to C1 is manually adjusted in order to avoid the obstacles.

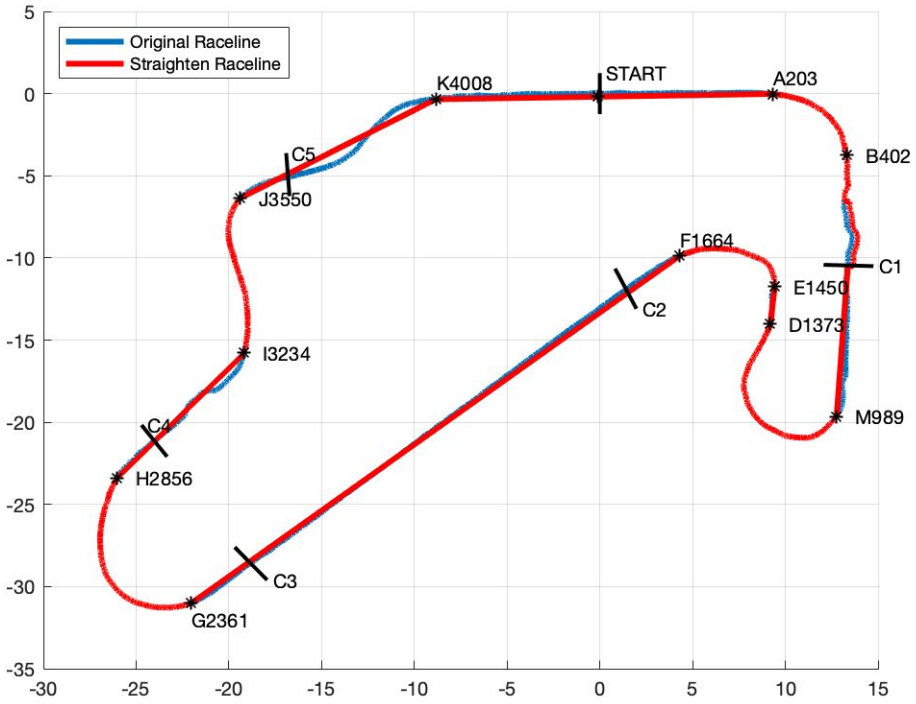

Fig.1 straightened raceline

1.3 Curve fitting

By calculating the average value of the closest 75 points for every waypoint, we can generate smoothed waypoints. Fig. 2 provides a close look at the first U-turn.

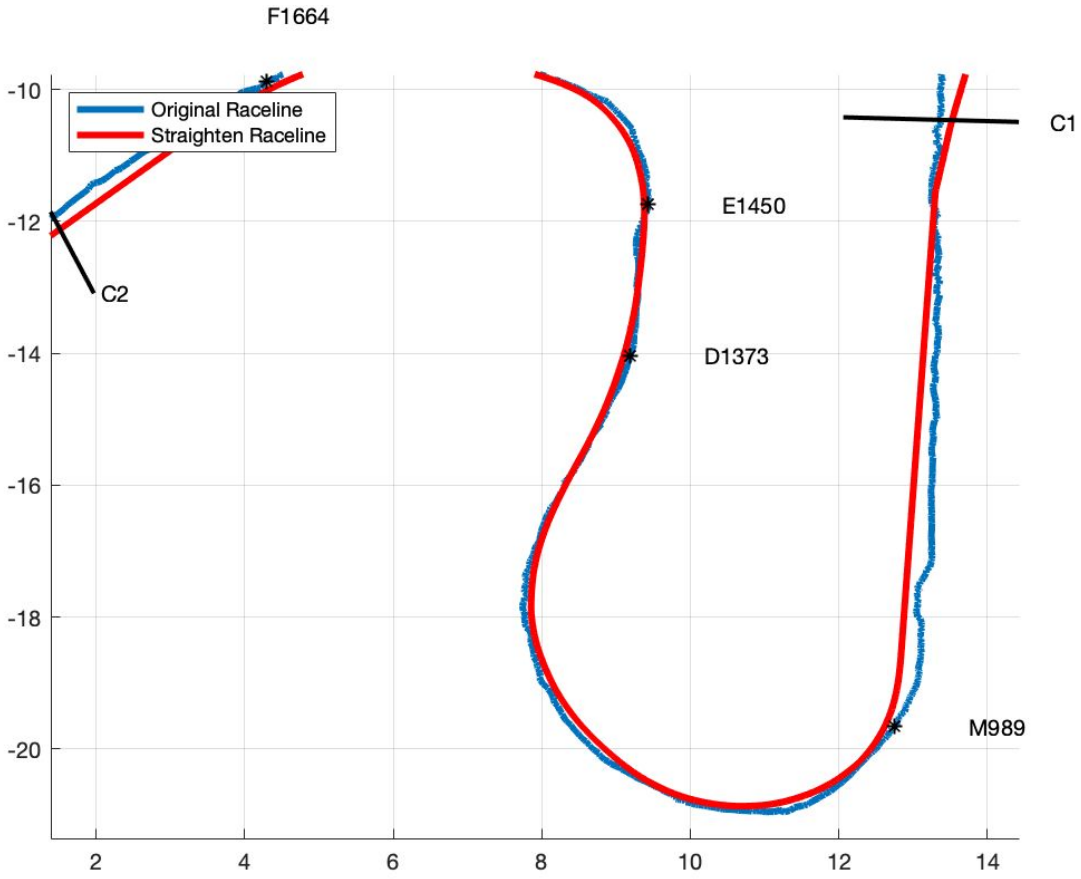

Fig.2 Smoothed straightened raceline

2. Pure Pursuit

2.1 Searching window `def reset_idx(self, cur_idx)`

To increase the desired point searching speed, We divide the waypoints at each checkpoint (C1, C2, C3, C4, C5 and Start). The checkpoint indexes corresponding to the waypoints are marked as chk_idx. When the searching index self.idx reaches chk_idx, self.idx starts at chk_idx. In this way, the past waypoints are ignored when searching for the desired point.

2.2 Desired point `def searchNew(self, cur_pos, path_points)`

Calculate the distance between the current car position and waypoint. Choose the closest waypoint within one and five times lookahead distance. If the distance greater than five times lookahead distance, the searching loop breaks. When self.idx reaches the total number of waypoints, self.idx is reset for each lap. Namely, self.idx starts from zero for every lap.

2.3 Adjusting lookahead distance and velocity `def update_lookahead(self, angle)`

We use a linear dynamic model for lookahead distance and velocity, which is controlled by the steering angle. Basically, for sharp turns, the lookahead distance and velocity are small; for straight lines, the values are large. By testing the car running in Rviz, we find the best maximum, minimum and increment values for lookahead distance and velocity.

2.4 Local speed control on the straight line `def velocity_adjust(self)`

There are two straight lines, K-A (18m) and F-G (33m), on the track that we care about. In order to make a smooth turn on the curves, the maximum speed (6.5m/s) is not full speed (10m/s) of the car model. We believe that locally applying full speed on the straight lines will save 2-3 seconds per lap. By overwriting the maximum speed at the start of the straight line (F and K) and linearly decreasing the maximum speed approaching the end of the straight line (G and A), the speed should transit smoothly. However, the breaking part is not in the best performance so we didn't use local speed control at the end. One possible hypothesis is the computation is not fast enough to adjust the maximum speed when the car is breaking.

2.5 Pure pursuit model `def PP_planner(self, cur_pose)`

When the current position (cur_pos) of the car in the map frame is changed and received, starting searches for the desired point (`searchNew`). Also, the searching index self.idx is reset for the current searching window. By calculating the distance between cur_pos and desired point, the curvature (kappa) and corresponding steering angle (angle) can be found. Based on the steering angle, the velocity can be calculated by `update_lookahead`. In order to implement the local speed control, before calculating the velocity, the maximum velocity is

updated by local speed controller `velocity_adjust`. Since we decide not to fulfill local speed control, this step is passed.

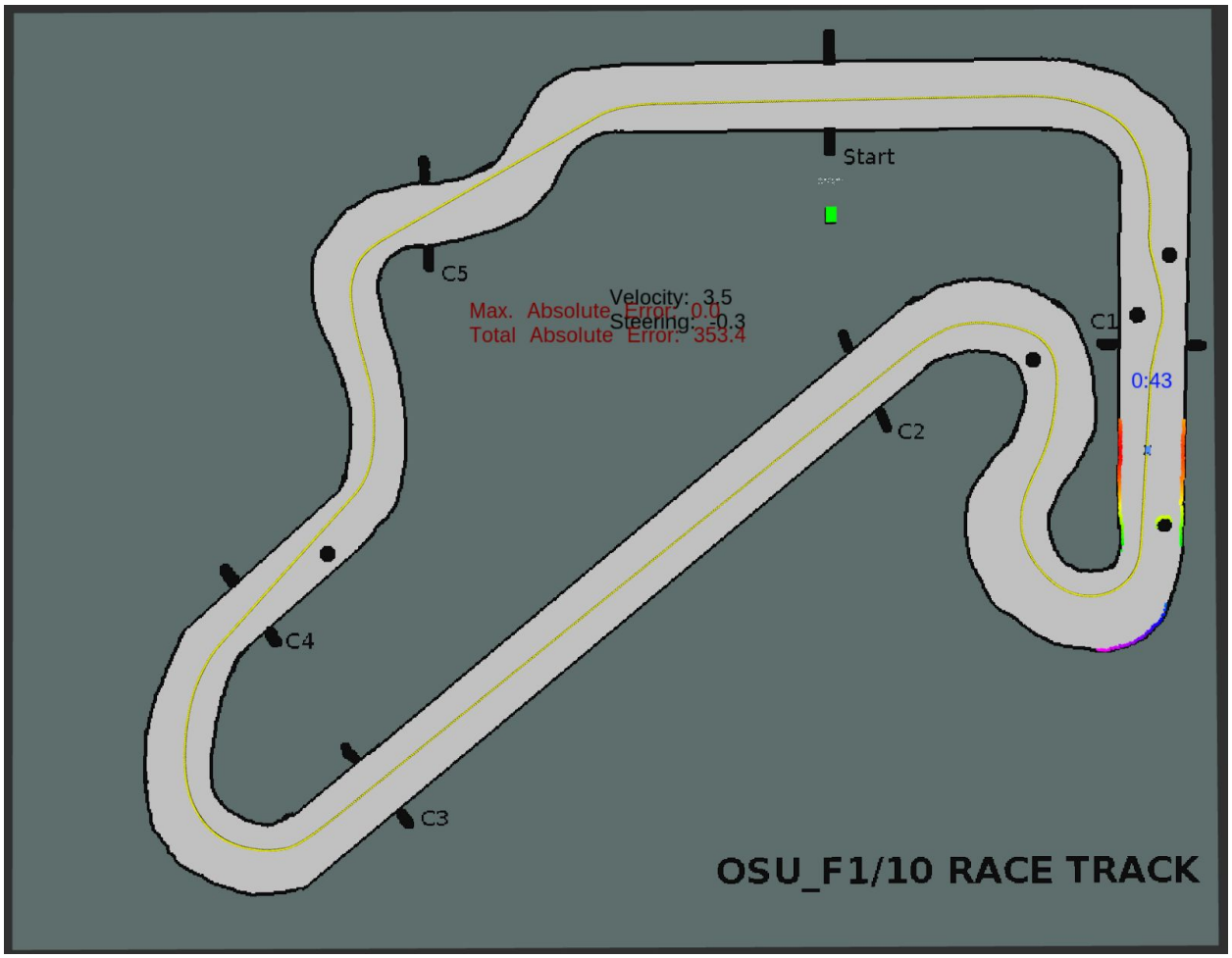

Fig. 3 Processed waypoint overview

3. Tracking Quality

3.1 Find two closest points

Since the tracking error is not time sensitive, we simply search the entire waypoints and find the closest two points (p1 and p2) from the location of the car (cur_point).

3.2 Tracking Error

By now, there are three points, p1, p2 and cur_point. First, calculate the angle between line p1 cur_point and line p1 p2 using cosine law. Then, calculate the length of line p1 cur_point.

Finally, The distance from cur_point to line p1 p2 can be calculated. This distance is the tracking error.

4. Bi-Directional RRT*

4.1 Waypoints calculation - Methodology

- Convert the map to a numpy matrix.
- Convert the (start, goal) map coordinates (cartesian) to the corresponding grid cells in the matrix using the YAML file.
- Construct the trees on both ends using regular RRT* by sampling uniformly throughout the space.
- Attempt to connect nodes of both the trees by a common node.
- Convert the nodes to the corresponding cartesian waypoints.
- Interpolate the waypoints to increase the population density.
- Apply moving average filter to smooth the waypoints.

4.2 RRT* Pseudo Code

```
Rad = r
G(V,E) //Graph containing edges and vertices
For itr in range(0…n)
   Xnew = RandomPosition()
   If Obstacle(Xnew) == True, try again
   Xnearest = Nearest(G(V,E),Xnew)
   Cost(Xnew) = Distance(Xnew,Xnearest)
   Xbest, Xneighbors = findNeighbors(G(V,E),Xnew,Rad)
   Link = Chain(Xnew,Xbest)
   For x’ in Xneighbors
      If Cost(Xnew) + Distance(Xnew,x’) < Cost(x’)
         Cost(x’) = Cost(Xnew)+Distance(Xnew,x’)
         Parent(x’) = Xnew
         G += {Xnew,x’}
   G += Link
Return G
```

4.3 Key points

- Euclidean distance was used to compute the nearest neighbor.
- Rewiring of the tree was performed by choosing the best parent, which is done by computing the node that has the best total cost so far. This rewiring process occurring for every node inserted into the tree, guaranteeing the best path found so far
- Two different sampling approaches were attempted: 1) Uniformly sampling all over the map, 2) Sampling in the neighborhood of the nodes that have been added to the tree. No method was found to be computationally advantageous that the other.
- Before adding a node to the tree, we ensure that the path is traversable by the car by adding a clearance region around the node and checking if it's obstacle-free.

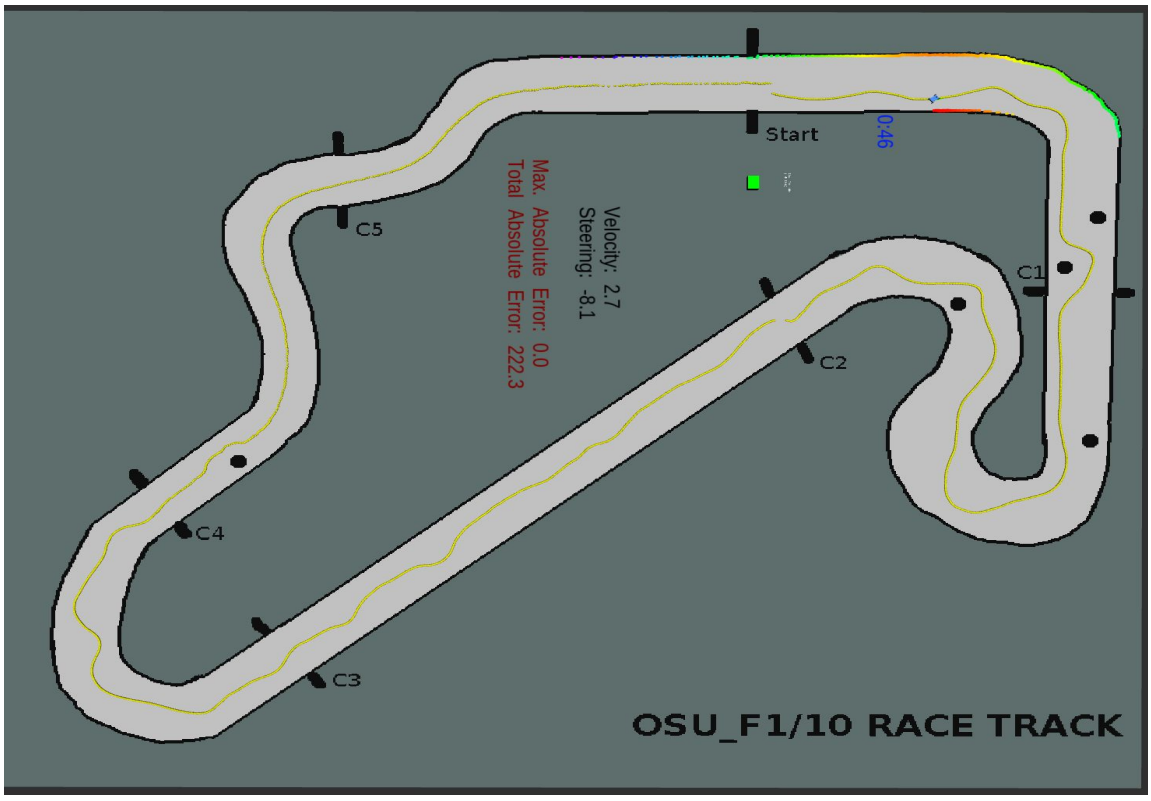

Fig.4 Waypoint from bi-directional RRT* overview

Our team chose bi-directional RRT* as it finds the potential optimal shortest possible path to the goal, or in our case the desired (target) point, as the number of nodes approaches infinity. This version of RRT* is bi-directional as it explores the tree from both start and goal node and expands in both directions.

To start, we get the closest node using the nearest neighbor function, which returns the closest node to the target node. Then, we check if the distance between the target node and the nearest node is within a certain radius, which we have determined to be 10 cells. If this check is passed, we then check if there is an obstacle between the closest node and the target node. If this check returns true, meaning there is no obstacle, we continue by sending the closest node to the chooseParent function. chooseParent will return the closest parent node to the target node if it is closer to the target than the current closest node. The final closest node is then added to the list of waypoints within the path. These waypoints are interpolated to get a smoother representation of the path. This process is repeated for each area between the checkpoints in order to get the best path in all locations within the map.

5. Final Race Approach Conclusion

Through testing both methods for determining waypoints, the most efficient and best performing approach was to pre-produce the waypoints using "center" wall following. The pre-produced waypoints method gave a faster lap time, faster velocity reached, and an overall smoother path, so it was the clear choice for the final race. From there, our group fine-tuned our parameters to get the fastest performance with no collisions and higher stability. The final parameters were determined to be a minimum velocity of 3.0, maximum velocity of 6.75, and a lookahead distance of 2.08. Our final race statistics gave us the fastest lap time to be 26.37 seconds over 11 laps within the 5 minutes allotted.