TECHNISCHE UNIVERSITÄT WIEN

Autonomous Racing Cars
191.119 (VU 4,0) Semester: 2022S

Pure Pursuit and Race Lines

2022-05-19

Student data

Matrikelnummer	Firstname	Lastname
12134689	Danilo	Castiglia
12135191	Artur	Chaves
01613004	Severin	Jäger
01552500	Johannes	Windischbauer

Transparency of contribution

- **Danilo Castiglia**: Pure pursuit and dynamic velocity
- **Artur Chaves**: Path visualization and evaluation
- **Severin Jäger**: Path planning implementation, gradient-based planning
- **Johannes Windischbauer**: Trajectory visualization, debugging

The finals section on the upcoming labs was written after a meeting in which the content was discussed among all team members.

Note

This page will not be shared with other teams. All following pages and the submitted source code archive will be shared in TUWEL after the submission deadline. Do not include personal data on subsequent pages.

Autonomous Racing Cars
191.119 (VU 4,0) Semester: 2022S

Pure Pursuit and Race Lines

2022-05-19

Team data

Team number: 2

1 Paths on maps

1.1 Implementation

a.) We implemented the path planning algorithm lined out in the assignment sheet. Firstly, the algorithm computes the driveable area and adds some safety foam around the obstacles. Then, it assigns each pixel a distance to the starting line starting at the finish. This procedure is visualised in Figure 4. In a next step, chooses the next pixel based on the lowest distance in the current pixels neighbourhood starting at the start position. Finally, the created path is sampled and published as ROS message. Listing 1 shows the corresponding source code.

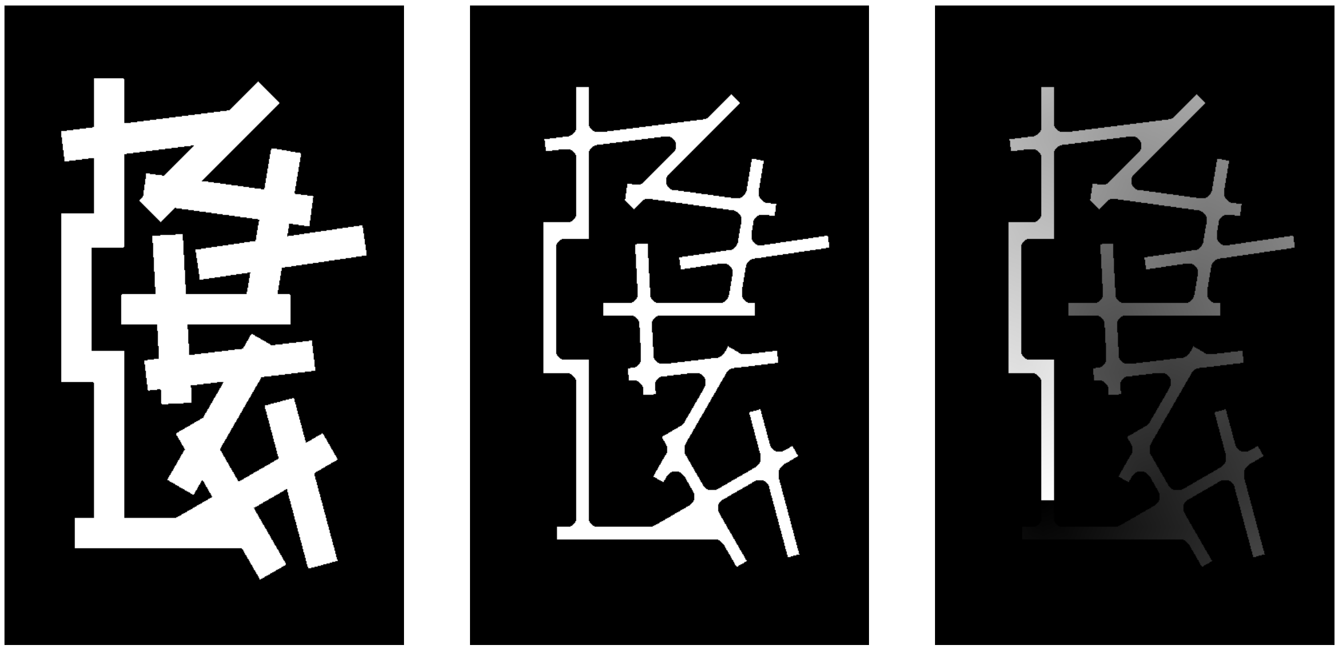

(a) Driveable area (b) Driveable area with safety foam (c) Distances (normalised)

Figure 1: Planner operation for the bunch_of_sticks map with `wall_distance`$= 0.6\,\mathrm{m}$

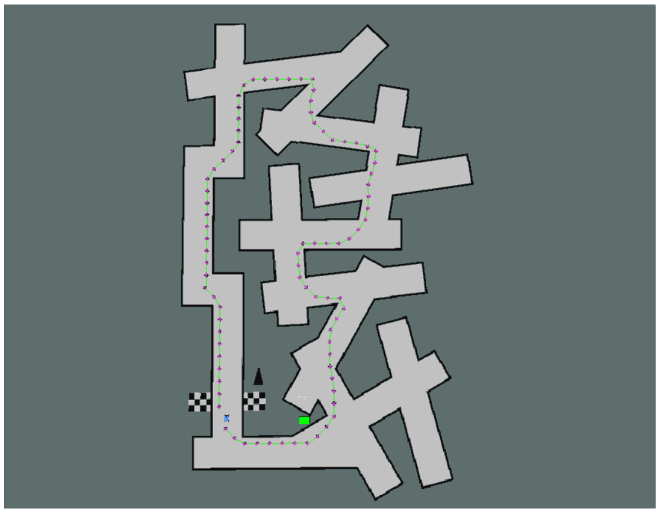
Our implementation has the following parameters:

- `wall_distance`: Radius of the safety foam (in meters).
- `waypoint_distance`: Distance between published waypoints (in meters).

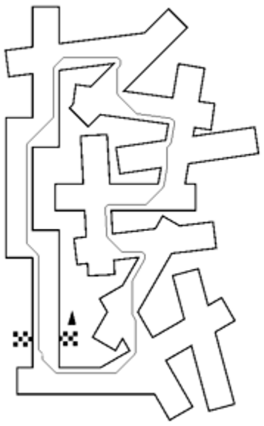
- `map_name`: Name of the map. Used for creating debugging images.
- `start_line_step`: Step size when scanning different starting line crossings
- `use_gradient_descent`: Enables the advanced path planning approach outlined in Section 2.2.b.

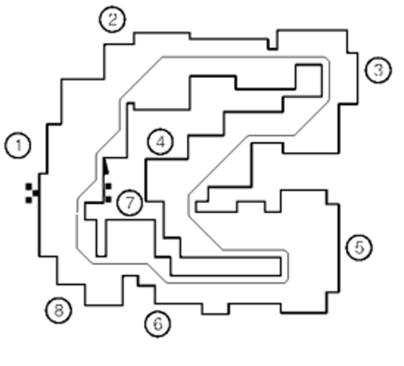
One major challenge in creating nice paths was the starting line. As the car always starts at the center of the line (at least in the simulator setup) and the shortest path to the line is not necessarily in the middle, the paths might not be closed shapes. This however leads to unsmooth driving around the line. To avoid this, we initially assigned a distance of 0 only to the middle of the line and calculated regular distances for the other line pixels. This however constrained the line crossing to the starting line center which is not always the shortest path. Therefore, we sweep over different starting positions and calculate the shortest path for each position. Eventually, we use the line crossing with the shortest lap length as the desired line crossing. The only disadvantage of this approach is the unsteady trajectory at the start. We alleviate this by publishing the first waypoint not right at the starting line but after one waypoint distance.

b.) The visualization that the algorithm is calculating can be seen in the Figure.

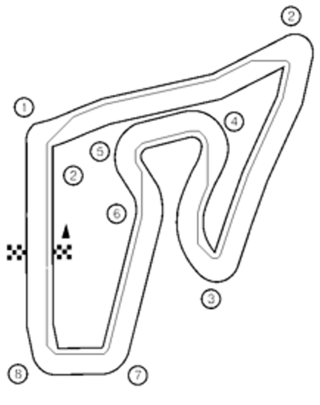

(a) Visualization of the path created

c.) The maps with the rendered path can be seen in the following images.


(a) bunch_of_sticks

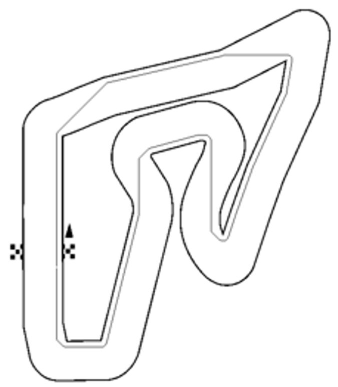


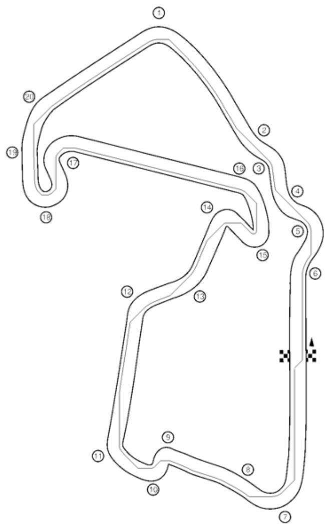
(b) train_pile_of_blocks



(c) f1_aut

Figure 3: Render of the path


(a) f1_aut_wide


(b) f1_gbr

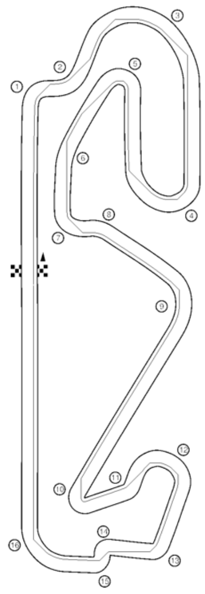
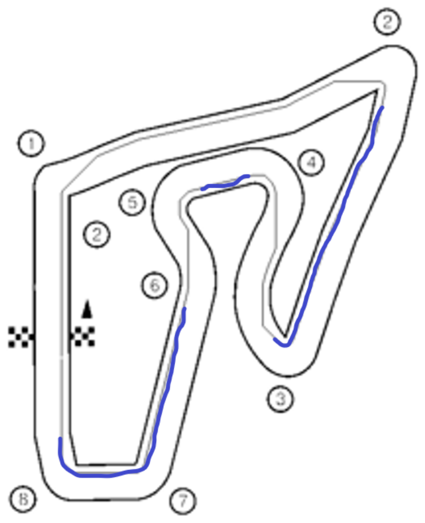

(c) f1_esp

Figure 4: Render of the path

d.) For the task 1.1.c. was used a n_{safety} of 0.4 meters, and that value was enough to create a path in all the tracks. But in some sharps turns like in f1_aut that n_{safety} could be increased to get the car to run more in the middle of the track getting better trajectory in some turns.

e.) From the task 1.1.c it was chosen the f1_aut to analyze. The path after the first turn looks almost optimal to approach turn 2, but when exiting turn 2 the path stays in the inside of the track which is not optimal to keep as much speed after the turn. when going between 2 an 3 still stays in the inside which when compared to the racing line of a professional diver is not the best option to make the turn 3, here the path could be optimized, different exit from turn 2 to keep the car in the outer part of the track to get a better turn 3. between turn 3 and 4 the path is good not much to change. Before turn 5 to get a more optimal path, it should go more to the middle of the track to get more speed through this turn. Then the last part of the track is where it should be more optimized, a professional driver would go in this sequences of turns more to the outside of the track and only go to the inside to make the turn, is a fast sequence of turns and speed is key. In this circuit some pilots have problems with track limits in turn 8 due to be better to go almost outside of the track in the outer part to get the most speed to the straight.



(a) Map with the zones that need attention

f.) There is differences between the shortest path and the optimal path. The shortest path tries to go on the inside part of the track all the time and in many cases that is not the best approach. Since the car is as mass and is moving, forces are going to be applied in the car, so is not optimal for the car to always stay in the inside of the track, because to keep that path it cannot be as fast. So the optimal path is the one that can keep the most speed in the car during the hole lap, so sometimes the car needs to cross from one side of the track to the other, making the path longer, to get a better entry in the turns to keep the momentum and have a higher speed through them. So with this in mind sometimes the shorter path compromises the lap times due to the capabilities of the car to follow it, because in a perfect world the shorter path would be the best.

g.) From the path that was obtain in task 1.1 e) and the characterisation made, it can be formulated some variables: path distance and the curve radius that the car can make the turn (because of the forces applied in the car). This variables will be related with each other. The main goal is to get the shortest path with the best curve radius for the car, which in some cases the to get the curve radius the path can drift from the shortest path, the same happens the other way around. Since the objective is to get the best time possible sometimes compromising one variable to the other can be beneficial.

h.) The algorithm can be improved by not only considering the shortest distance but also other optimisation criteria like the curve radius. Additionally, some smoothing might help creating steady trajectories. One way to implement these improvements is our gradient-descent based approach outlined in 2.2.b. – which however does not consider curve radii yet.

2 Pure Pursuit

2.1 Implementation

a.) We implemented the pure pursuit algorithm as described in the assignment sheet. The source code of our implementation is given in Listing 2.

b.) To visualize the goal point we used the Marker visualization message already provided in the template. We are publishing the message every time we are calculating a new point. A video of a visualization with a relatively small look-ahead distance for the goal point can be seen in the additional upload. The visualization is turned off by default and can be enabled by setting the rosparam `visualization:=true`.

c.) For this task we introduced two parameters, the `log_output` and the `log_output_length`. The first one is a boolean parameter that defines if the log output will be published to the info log level and the output length defines n for the output, so every n-th log message gets published. A bigger log output length means less messages are being published.

d.) For plotting the ground truth trajectory we decided to work with the Marker visualization message again instead of the MarkerArray. The Marker message offers a type for a list of spheres and a points array, which improves the visualization performance in RViz according to the specification. The only constraint being that the individual spheres have to have the same size, which is not a problem for this task. The trajectory publisher then publishes a new ground pose sphere every n-th log message resulting in a queue. By adding the `queue_length` parameter we can decide how many points the path should contain to not overload the system with hundreds of thousand trajectory points.

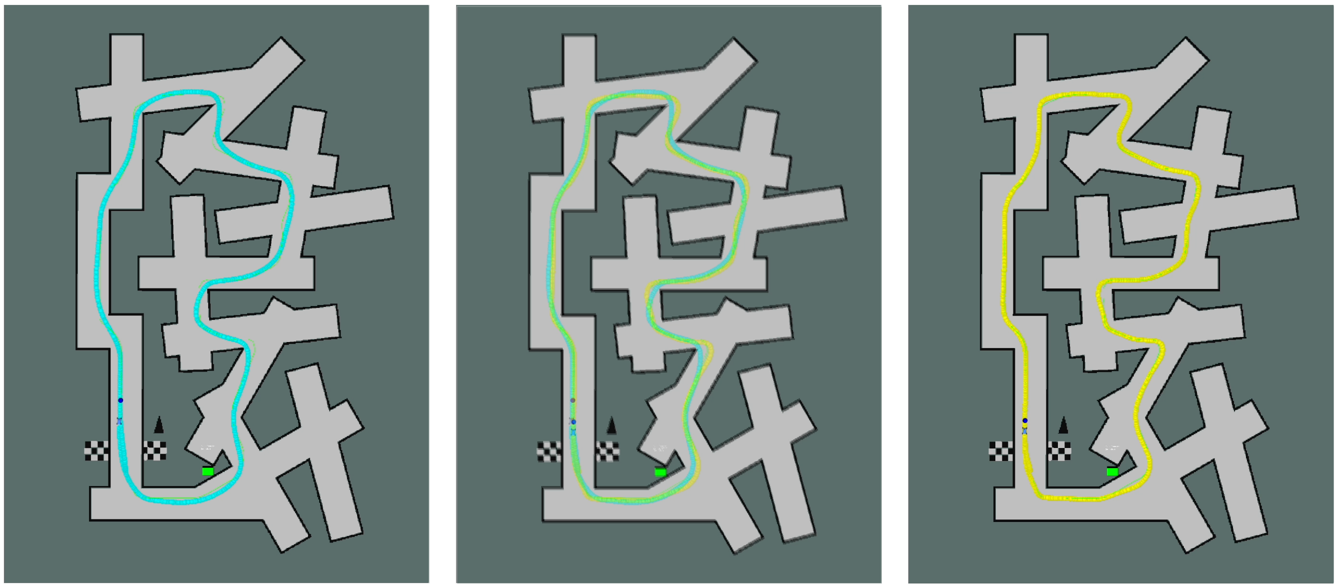

(a) Path with a look-ahead distance of 1.5 (b) Overlay of the paths of the figure left and right (c) Path with a look-ahead distance of 0.7

Figure 6: Paths visualized

e.) In our implementation the look-ahead distance L has a great impact on the smoothness of the trajectory our car produces. A shorter look-ahead distance causes oscillation because small deviations from the path result in big steering angles causing the car to drift and further miss the planned path. An overlay of the path with different look-ahead distances, namely 0.7m and 1.5m can be seen in figure 6.

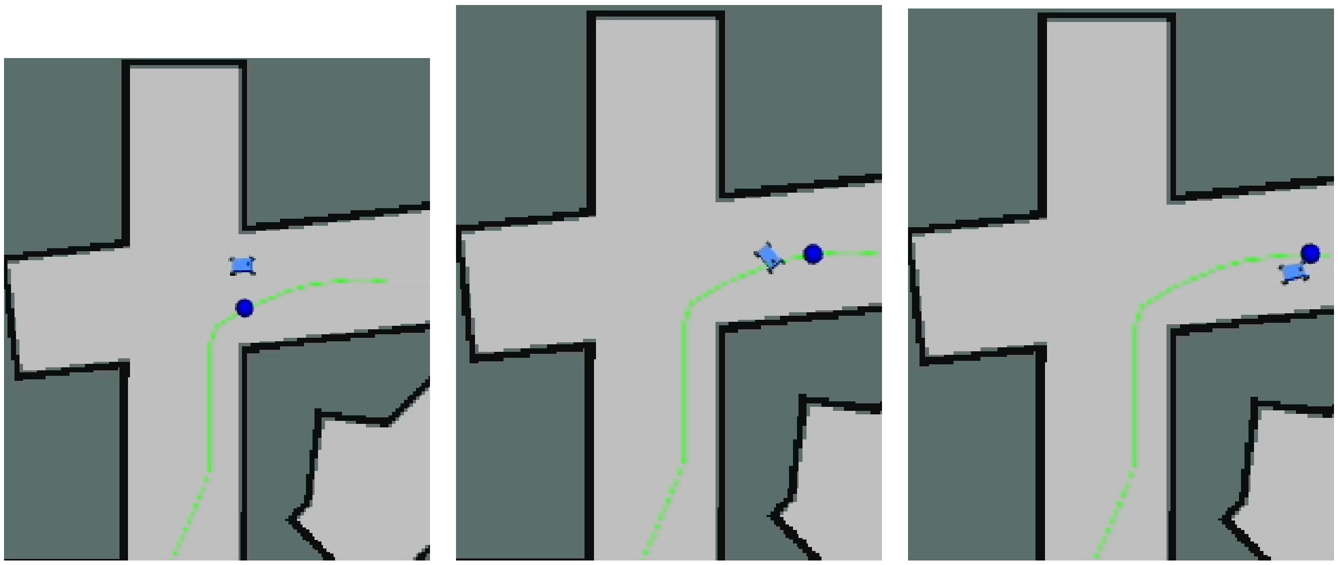


Figure 7: Small look-ahead causes oscillation

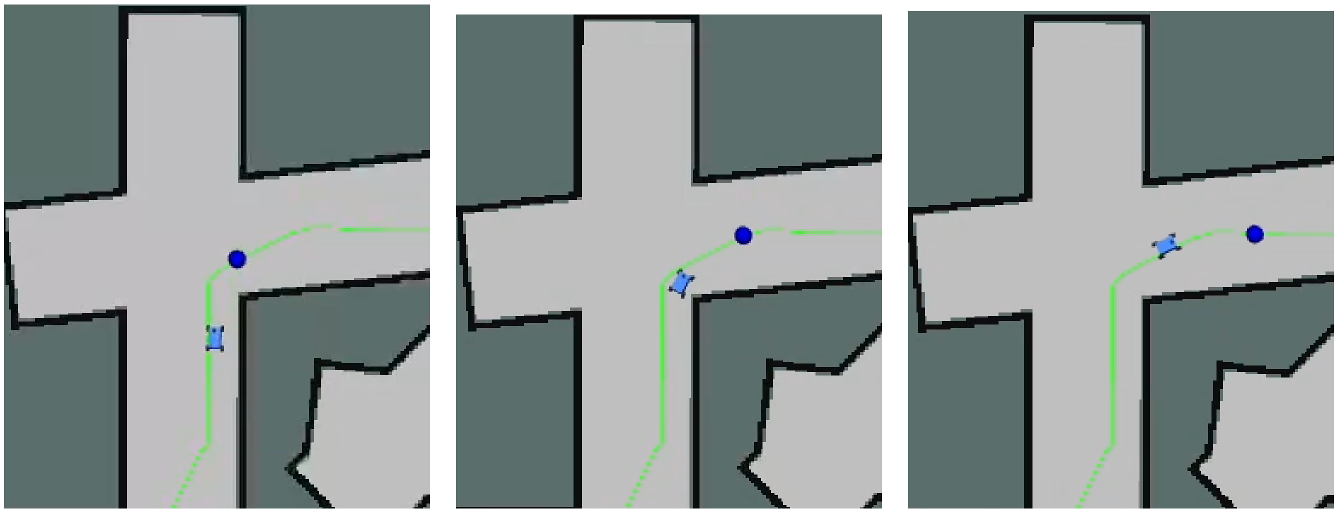


Figure 8: Larger look-ahead causes smoother steering and driving

However, larger look-ahead distance also come with the trade off that sharp corners cause crashes because the car tries to cut the corner, since the goal point is already further down the road.

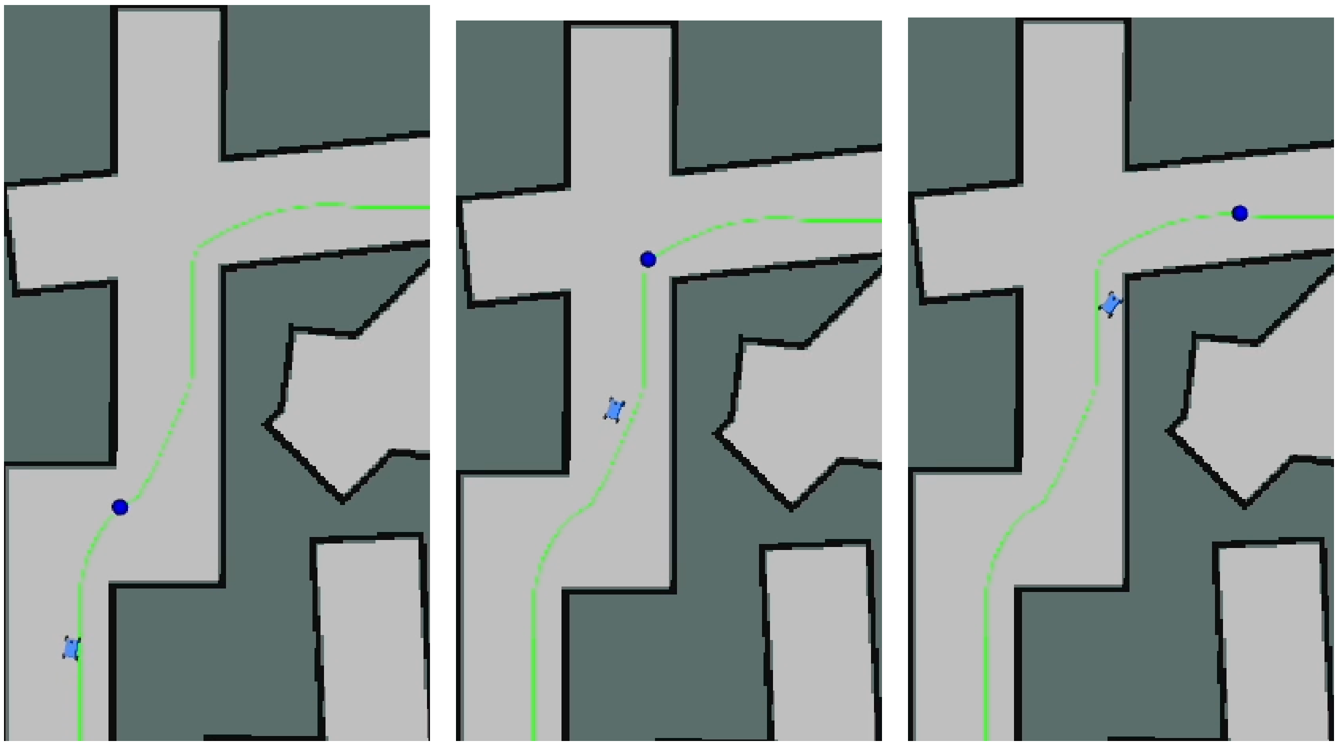


Figure 9: Look-ahead too far so the car tries to cut corners too aggressively

f.) As of writing this part of the write up we were able to tune our controller to be able to complete the course on the testbench in around 21 seconds.

2.2 Building a more advanced controller

a.) To improve our lap times, we implemented a very simple velocity heuristic which only considers the lidar distance directly in front of the car and multiplies it with a constant gain to obtain the desired velocity. The resulting value is clipped using a minimum and a maximum velocity. However, this is more or less a reactive method and it is likely favourable to consider the knowledge of upcoming laps from the planned path.

b.) As discussed in Section 1.1, the basic shortest path algorithm from the assignment sheet has some disadvantages. Firstly, it is limited to the neighbouring pixels which limits potential angles to multiples of 45°. Thus, it does not necessarily produce the shortest possible path. Furthermore, it tends to unsmooth trajectories like the start line crossing in Figure 10a. Such frequent steering manoeuvres can drastically limit the maximum achievable speed without losing control and are thus unfavourable for racing.

To alleviate the aforementioned problems, we introduced a new path planning algorithm based on the well-known gradient descent method. It reuses the distance map calculated by the basic algorithm but instead of selecting the best neighbouring pixel it computes a gradient map of the distances and follows the negative distance gradient. Furthermore, we do not only consider the current gradient orientation but also the last orientation and the orientation at the projected next step in a weighted way to provide smoother paths.

This updated algorithm has the following additional parameters:

- `gradient_step_size`: Step size for the position updates (in pixels). The step size is however automatically reduced if the destination is outside the driveable area.
- `gradient_weight_past`: Weight of the last gradient orientation.
- `gradient_weight_future`: Weight of the projected next gradient orientation.

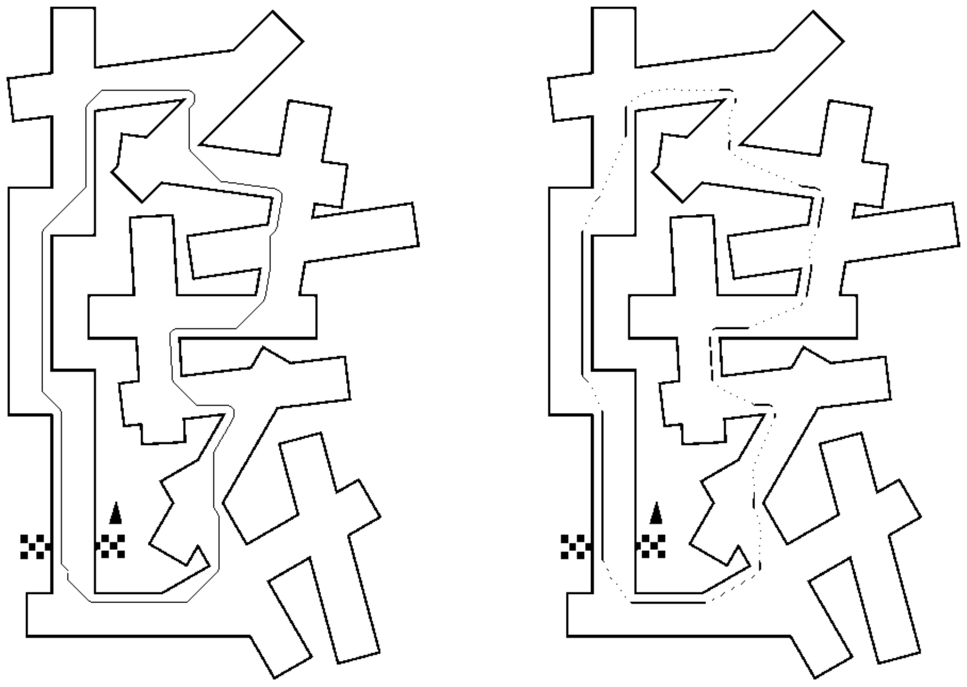
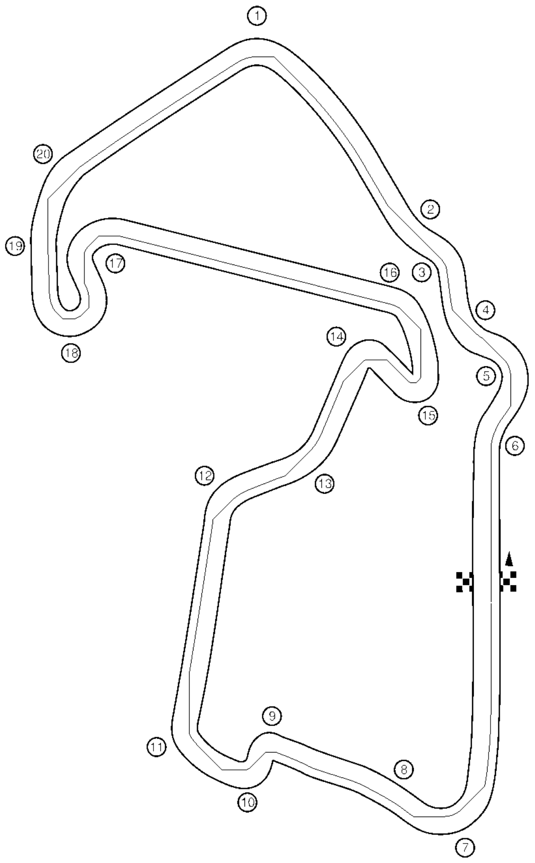

(a) Basic algorithm (b) Gradient-based planner

Figure 10: Planned paths in the bunch_of_sticks map

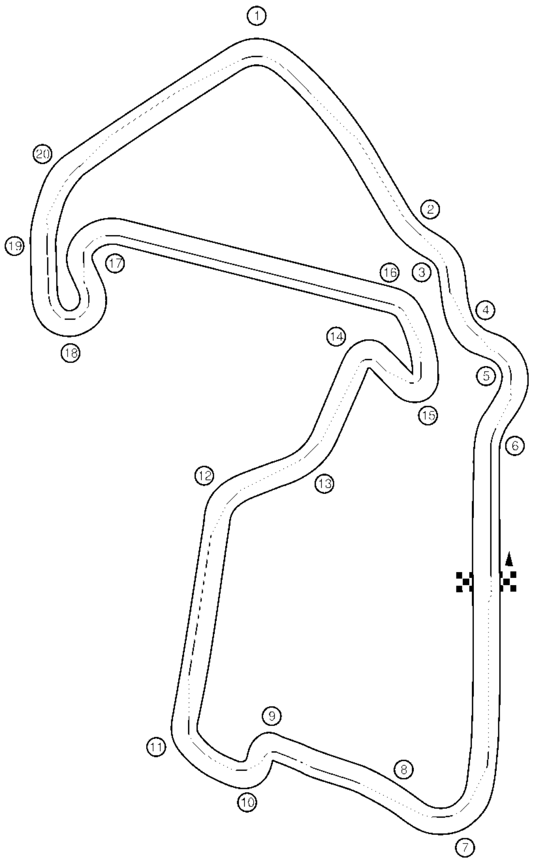
The results for the bunch_of_sticks map are shown in Figure 10 where the gradient-based path is shorter (70.1 m vs. 71.6 m) and clearly smoother. The larger step size is directly visible as the path is dotted in sections where it does not follow the border of the driveable area.

Similar observations can be made for the f1_gbr map, s. Figure 11. Again, the path is shorter with gradient-based planning (196.9 m vs. 199.1 m). Additionally, our improved algorithm causes much smoother curves, which are for instance apparent in Figure 11b between curves 12 and 16. Interestingly, our smoothing causes the path to deviate from the shortest route in some situations. The section between curve 1 and 20 in Figure 11b is a good example for this behaviour. Here, the algorithm calculates a slightly wider path which however again introduces less aprubt steering and enables therefore higher velocities.

c.) (not covered)



(a) Basic algorithm



(b) Gradient-based planner

Figure 11: Planned paths in the f1_gbr map

3 Proposal for further work

a.) For lab 7 we aim at optimising our existing algorithms. We will base our work on our existing implementations of the disparity extender, the global planner, and the pure pursuit algorithm.

One crucial component for racing is the velocity calculation. We want to find a way to compute the fastest speed based on the available sensor information for both approaches. To assess the algorithms, we will extend the simulator with a lap time computation and plot the speed over time to look for optimisation potential. To minimise the simulation-to-reality gap we will try to measure or estimate the simulator parameters as good as possible.

- For the disparity extender, we already have a velocity mapping based on the stretch length. We will try to improve this by finding a reasonable nonlinear mapping, potentially based on the expected (or measured) braking behaviour.
- For the planning-based navigation we want to consider not only the distance towards an obstacle ahead, but mainly the knowledge about upcoming curve radii for velocity calculation.

After tuning both algorithms, we should be able to compare them on a variety of simulator maps (as well as on-site maps using SLAM). This helps us to decide how to proceed in lab 8 as we think disparity extender is already a very nice racing algorithm and we are not sure how much we can actually gain from global planning – especially in the light of mapping and localisation imperfections.

To assess these imperfections we want to try our path planning algorithms on the hardware car as fast as possible to be able to mitigate the simulation-to-reality gap quickly.

Furthermore, we think that better path planning can help to improve the race performance. We plan to extend our planner beyond the improvements outlined in Section 2.2.b. by considering not only the distance towards the finish line, but also maximising curve radii. We want to follow the gradient of the (weighted) superposition of both metrics to improve our paths. To do so, we need to compute the curve radius for each pixel after every position update in the planning phase. This can be based on lookahead trajectories (which have to be as long as an average curve on the map). At the moment, the trajectories are mainly smoothed out by the lookahead distance of the pure pursuit algorithm. However, this does not necessarily use the space on the map efficiently and might cause collisions. Instead, a better planner can create smoother paths (which make pure pursuit easier) and increase the possible velocities. To evaluate and tune this algorithm, we want to consider again the lap times as well as the path lengths.

In lab 8 we plan to develop methods that allow us to cope with map uncertainty and obstacles (mainly other cars). This can be either based on local replanning or combinations of reactive and planning-based algorithms. However, to define this work in further detail we need additional data from the hardware car as we cannot assess the performance of planning-based navigation on the real car at the moment.

4 Appendix

Listing 1: Path Planner Node

```
#!/usr/bin/env python3
from __future__ import print_function
import sys
import math
import numpy as np
import os
import tf

#ROS Imports
import rospy
import geometry_msgs
from nav_msgs.msg import OccupancyGrid
from geometry_msgs.msg import PoseStamped
from nav_msgs.msg import Path

# scikit-image for image operations
from skimage import io, morphology, img_as_ubyte, __version__

# avoid problems with python2
MAX_FLOAT = 99999.9


class planner:
    def __init__(self):
        rospy.loginfo("Hello␣from␣planner␣node!")
        rospy.loginfo("scikit-image␣version␣" + __version__)

        # Topics & Subscriptions, Publishers
        map_topic = '/map'
        path_topic = '/path'

        self.map_sub = rospy.Subscriber(map_topic, OccupancyGrid, self.
           ↪ map_callback, queue_size=1)
        self.path_pub = rospy.Publisher(path_topic, Path, queue_size=10, latch=
           ↪ True)

        # Parameters
        self.OCCUPIED_THRESHOLD = 0.65  # map pixels are considered occupied if
           ↪  larger than this value
        self.SAFETY_DISTANCE = rospy.get_param('/planner/wall_distance', 0.6)
           ↪ # safety foam radius (m)
        self.WAYPOINT_DISTANCE = rospy.get_param('/planner/waypoint_distance',
           ↪ 0.1)  # distance between waypoints (m)
        self.MAP_NAME = rospy.get_param('/planner/map_name', "")  # name for
           ↪ the exported map
        self.START_LINE_STEP = rospy.get_param('/planner/start_line_step', 4)
           ↪ # in pixels

        self.STEP_SIZE = rospy.get_param('/planner/gradient_step_size', 5 * np.
           ↪ sqrt(2))
        self.WEIGHT_PAST = rospy.get_param('/planner/gradient_weight_past',
            ↪ 0.35)
        self.WEIGHT_FUTURE = rospy.get_param('/planner/gradient_weight_future',
            ↪  0.15)
        self.MAX_REASONABLE_ANGLE = math.radians(50)

        # False: basic approach as in assignment sheet
        # True:  gradient descent based path calculation with larger step size
        self.USE_GRADIENT_DESCENT = rospy.get_param('/planner/
            ↪ use_gradient_descent', True)

        # Path for saving maps
        self.savepath = os.path.dirname(os.path.realpath(__file__)) + "/../maps
            ↪ "
        if not os.path.exists(self.savepath):
            os.makedirs(self.savepath)

    def map_callback(self, data):
        """
        Process the map to pre-compute the path to follow and publish it
        """
        rospy.loginfo("map␣info␣from␣OccupancyGrid␣message:\n%s", data.info)

        self.shape = (data.info.width, data.info.height)
        self.start_position = (data.info.origin.position.x,
                               data.info.origin.position.y)
        self.resolution = data.info.resolution
        self.start_pixel = (int(-self.start_position[0]/self.resolution),
                            int(-self.start_position[1]/self.resolution))

        # in pixels
        self.WAYPOINT_DISTANCE = rospy.get_param('/planner/waypoint_distance',
            ↪ 0.8)/self.resolution

        map = self.preprocess_map(data)

        driveable_area = self.get_driveable_area(data, map)
        self.save_map(driveable_area, "1_drivable_area")

        driveable_area = self.add_safety_foam(driveable_area)
        self.save_map(driveable_area, "2_drivable_area_safety")

        shortest_lap = MAX_FLOAT

        # plan path for different starting positions, consider the shortest
            ↪ round-trip
        for start_x in range(self.start_line_left+1, self.start_line_right,
            ↪ self.START_LINE_STEP):

            path_msg = Path()
            path_msg.header.frame_id = "map"
            path_msg.header.stamp = rospy.get_rostime()

            distances = self.get_distances(driveable_area, start_x)
            path, path_msg, lap_length = self.calculate_path(map, distances,
               ↪ path_msg, start_x=start_x)

            # lap_length = distances[start_x, self.start_pixel[1] + 2]
            rospy.loginfo("Start␣@" + str(start_x) + ":␣track␣length␣" + str(
               ↪ lap_length))
            if lap_length <= shortest_lap:
                shortest_lap = lap_length
                self.start_x = start_x
                self.distances = distances
                self.path = path
                self.path_msg = path_msg

        self.lap_length = shortest_lap
        rospy.loginfo("Shortest␣Lap:␣Start␣@" + str(self.start_x) + ":␣track␣
           ↪ length␣" + str(self.lap_length))

        distances_print = self.distances.copy()
        distances_print[driveable_area == False] = 0.0  # remove high values
           ↪ outside driveable area
        self.save_map(distances_print/np.max(distances_print), "3_distances")

        self.path_pub.publish(self.path_msg)
        self.save_map(self.path, "4_path")

    def get_distances(self, driveable_area, start_x):
        distances = np.full(self.shape, MAX_FLOAT, dtype=float)
        done_map = (~(driveable_area)).copy()

        # Mark starting line as done, distance 0 for start position
        y = self.start_pixel[1]
        for x in range(self.start_line_left, self.start_line_right+1):
            done_map[x, y] = True
            distances[x, y] =  abs(x-start_x)

        queue = []
        queue.append((self.start_pixel[0], self.start_pixel[1]-1))

        queued_map = done_map.copy()
        while queue:
            (x, y) = queue.pop(0)
            min_dist, min_x, min_y = self.get_min_dist_neighbour(done_map,
               ↪ distances, x, y)
            done_map[x, y] = True
            new_dist = np.linalg.norm(np.array([x, y]) - np.array([min_x, min_y
               ↪ ])) + min_dist
            if new_dist < distances[x, y]:
                distances[x, y] = new_dist
            for [step_x, step_y] in [[0, 1],[0, -1],[1, 0],[-1, 0],[1, 1],[-1,
               ↪ 1],[1, -1],[-1, -1]]:
                new_x = x+step_x
                new_y = y+step_y
                if not done_map[new_x, new_y] and not queued_map[new_x, new_y]:
```

```
                done_map[new_x, new_y] = True
                queue.append((new_x, new_y))
        return distances

    def preprocess_map(self, data):
        map_data = np.asarray(data.data).reshape((data.info.width, data.info.
            ↪ height)) # parse map data into 2D numpy array
        map_normalized = map_data / np.amax(map_data.flatten()) # normalize map
        map_binary = map_normalized < (self.OCCUPIED_THRESHOLD) # make binary
            ↪ occupancy map
        return map_binary

    def get_driveable_area(self, data, map_binary):
        driveable_area = morphology.flood_fill(
            image = 1*map_binary,
            seed_point = self.start_pixel,
            new_value = -1,
        )
        driveable_area = driveable_area < 0
        return driveable_area

    def add_safety_foam(self, driveable_area):
        # selem for scikit-image 16.2, new name: footprint
        binary_image = morphology.binary_erosion(driveable_area, selem=
            ↪ morphology.disk(radius=self.SAFETY_DISTANCE/self.resolution,
            ↪ dtype=bool))

        x = self.start_pixel[0]+1
        while binary_image[x, self.start_pixel[1]]:
            x = x + 1
        self.start_line_right = x-1

        x = self.start_pixel[0]-1
        while binary_image[x, self.start_pixel[1]]:
            x = x - 1
        self.start_line_left = x+1
        return binary_image

    def get_min_dist_neighbour(self, done_map, distances, x, y):
        min_dist = MAX_FLOAT
        for [step_x, step_y] in [[0, 1],[0, -1],[1, 0],[-1, 0],[1, 1],[-1,
            ↪ 1],[1, -1],[-1, -1]]:
            if done_map[x+step_x, y+step_y]:
                if distances[x+step_x, y+step_y] < min_dist:
                    min_dist = distances[x+step_x, y+step_y]
                    x_min = x+step_x
                    y_min = y+step_y
        return min_dist, x_min, y_min

    def calculate_path(self, map, distances, path_msg, start_x):
        # Start a little above the starting line as the line has distance 0
        # The pixels just above have distance 1 due to an imperfect distance
            ↪ calculation
        START_OFFSET = 2
```

```
x = start_x
y = self.start_pixel[1] + START_OFFSET
distance = distances[x, y]

path_length = 1.0 * START_OFFSET
path = map.copy()

best_x = -1
best_y = -1

# TODO
# curvature =  np.full(self.shape, MAX_FLOAT, dtype=float)
# no curvature initially

path[x,y] = 0.0
# Fist step: only steps up (or sideways) as first gradient points
   ↪ towards line
for [step_x, step_y] in [[0, 1],[1, 0],[-1, 0],[1, 1],[-1, 1]]:
    new_x = x + step_x
    new_y = y + step_y
    if distances[new_x, new_y] < distance:
        distance = distances[new_x, new_y]
        best_x = new_x
        best_y = new_y

if best_x == -1 and best_y == -1:
    rospy.logerr("No␣valid␣path␣found␣from␣this␣starting␣position.␣
       ↪ Consider␣increasing␣START_OFFSET.")
    exit(-1)

last_x = x
last_y = y
x = best_x
y = best_y
last_waypoint = np.array([x, y])

# Not broadcasting a position at the start reduces problems with a non-
   ↪ central start
# self.add_pose_to_path(path_msg, x, y, last_x=last_x, last_y=last_y)

if not self.USE_GRADIENT_DESCENT:
    while(distance > 0.0):
        path[x,y] = 0.0
        path_point = np.array([x, y])

        if np.linalg.norm(path_point - last_waypoint) >= self.
           ↪ WAYPOINT_DISTANCE:
            self.add_pose_to_path(path_msg, x, y, last_x=last_x, last_y
               ↪ =last_y)
            path_length += np.linalg.norm(path_point - last_waypoint)
            last_waypoint = path_point

        for [step_x, step_y] in [[0, 1], [0, -1],[1, 0],[-1, 0],[1,
           ↪ 1],[-1, 1],[1, -1],[-1, -1]]:
            new_x = x+step_x
            new_y = y+step_y
            if distances[new_x, new_y] < distance:
                distance = distances[new_x, new_y]
                best_x = new_x
                best_y = new_y
        last_x, last_y = x, y
        x, y = best_x, best_y

    # add final waypoint (to have a fair distance comparison)
    self.add_pose_to_path(path_msg, x, y, last_x=last_x, last_y=last_y)
    path_length += np.linalg.norm(path_point - last_waypoint)

else:  # USE_GRADIENT_DESCENT
    g_x, g_y = np.gradient(distances)
    orientation = np.arctan2(g_x, g_y)+np.pi

    # self.save_map((orientation+np.pi)/(np.max(orientation)* 2 * np.pi
       ↪ ), "5_orientation")
    dir = orientation[x, y]

    while(distance > self.STEP_SIZE-1):
        path[x,y] = 0.0
        path_point = np.array([x, y])

        if np.linalg.norm(path_point - last_waypoint) >= self.
           ↪ WAYPOINT_DISTANCE:
            self.add_pose_to_path(path_msg, x, y, orientation=
               ↪ orientation[x, y])
            path_length += np.linalg.norm(path_point - last_waypoint)
            last_waypoint = path_point

        # TODO implement curve radius function
        # LOOKAHEAD_DISTANCE = 5
        # DISTANCE_WEIGHT = 0.7
        # CURVATURE_WEIGHT = 1-DISTANCE_WEIGHT

        # get_reasonable_steps: return legal pixels (given position,
           ↪ step size, angular range, init dir)
        # get_reasonable_steps((x, y), STEP_SIZE, np.pi, last_direction
           ↪ )

        # last_direction = dir

        if np.abs(dir - orientation[x, y]) < self.MAX_REASONABLE_ANGLE
           ↪ and \
           np.abs(orientation[int(x + self.STEP_SIZE * np.sin(
              ↪ orientation[x,y])+0.5),
                 int(y + self.STEP_SIZE * np.cos(orientation[x,y])
                    ↪ +0.5)] - orientation[x, y]
                ) < self.MAX_REASONABLE_ANGLE:
            dir = self.WEIGHT_PAST * dir + \
                 (1-self.WEIGHT_PAST - self.WEIGHT_FUTURE) *
```

```
                             ↪ orientation[x, y] + \
                         self.WEIGHT_FUTURE * orientation[int(x + self.
                             ↪ STEP_SIZE * np.sin(orientation[x,y])+0.5), int(
                             ↪ y + self.STEP_SIZE * np.cos(orientation[x,y])
                             ↪ +0.5)]
                    next_step = self.STEP_SIZE
                else:
                    dir = orientation[x, y]
                    next_step = np.sqrt(2)

                new_x = int(x + next_step * np.sin(dir)+0.5)
                new_y = int(y + next_step * np.cos(dir)+0.5)

                # keep path in drivable area
                j = 1
                while distances[new_x, new_y] == MAX_FLOAT:
                    new_x = int(x + (next_step-j*np.sqrt(2)) * np.sin(dir)+0.5)
                    new_y = int(y + (next_step-j*np.sqrt(2)) * np.cos(dir)+0.5)
                    j = j + 1
                    if (next_step-j*np.sqrt(2)) <= 0 or (next_step-j*np.sqrt(2)
                        ↪ ) <= 0:
                        break

                last_x = x
                last_y = y
                x = new_x
                y = new_y
                distance = distances[x, y]

            # add final waypoint (to have a fair distance comparison)
            self.add_pose_to_path(path_msg, x, y, orientation=orientation[x, y
                ↪ ])
            path_length += np.linalg.norm(path_point - last_waypoint)
        return path, path_msg, path_length

    def add_pose_to_path(self, path_msg, x, y, last_x=-1, last_y=-1,
        ↪ orientation=0.0):
        pose_msg = PoseStamped()
        pose_msg.header.frame_id = "map"
        pose_msg.header.stamp = rospy.get_rostime()
        pose_msg.pose.position.x = self.resolution*y + self.start_position[0]
        pose_msg.pose.position.y = self.resolution*x + self.start_position[1]
        pose_msg.pose.position.z = 0.0

        if last_x == -1 and last_y == -1:
            orientation = orientation
        else:
            orientation = math.atan2(x-last_x, y-last_y)

        pose_msg.pose.orientation = geometry_msgs.msg.Quaternion(*tf.
            ↪ transformations.quaternion_from_euler(0.0, 0.0, orientation))
        path_msg.poses.append(pose_msg)

    def save_map(self, map, name=""):
        save_path = self.savepath + '/' +  self.MAP_NAME + "_" + name + '.png'
        map_image = np.rot90(np.flip(map, 0), 1) # flip and rotate for
           ↪ rendering as image in correct direction
        io.imsave(save_path, img_as_ubyte(map_image), check_contrast=False) #
           ↪ save image, just to show the content of the 2D array for debug
           ↪ purposes
        rospy.loginfo("map␣saved␣to␣%s", save_path)

def main(args):
    rospy.init_node("planner_node", anonymous=True)
    rfgs = planner()
    rospy.sleep(0.1)
    rospy.spin()

if __name__ == '__main__':
    main(sys.argv)
```

Listing 2: Pure Pursuit Node

```
#!/usr/bin/env python3
from __future__ import print_function
import sys
import math
import numpy as np

#ROS Imports
import rospy
import math
from sensor_msgs.msg import LaserScan
from ackermann_msgs.msg import AckermannDriveStamped, AckermannDrive
from geometry_msgs.msg import Point
from visualization_msgs.msg import Marker
from visualization_msgs.msg import MarkerArray
from nav_msgs.msg import OccupancyGrid
from nav_msgs.msg import Odometry
from geometry_msgs.msg import PoseStamped
from nav_msgs.msg import Path
import tf2_ros
import tf2_geometry_msgs

from skimage import io, morphology, img_as_ubyte

# Parameters of Pure Pursuit
LOOKAHEAD_DISTANCE = rospy.get_param('/pure_pursuit/lookahead_distance', 1.2)
STEERING_GAIN = rospy.get_param('/pure_pursuit/steering_gain', 0.5)
BASIC_VELOCITY = rospy.get_param('/pure_pursuit/basic_velocity', False)
VELOCITY = 2
MAX_SPEED = rospy.get_param('/pure_pursuit/max_speed', 5)  # m/s  (only without
   ↪  basic velocity)
MIN_SPEED = rospy.get_param('/pure_pursuit/min_speed', 1.75)  # m/s  (only
   ↪ without basic velocity)
VELOCITY_GAIN = rospy.get_param('/pure_pursuit/velocity_gain', 1.25)
# Visualization parameters
VISUALIZATION = rospy.get_param('/pure_pursuit/visualization', False)
LOG_OUTPUT = rospy.get_param('/pure_pursuit/log_output', True)
```

```
LOG_OUTPUT_LENGTH = rospy.get_param('/pure_pursuit/log_output_length', 100) #
    ↪ only every 100th message
QUEUE_LENGTH = rospy.get_param('/pure_pursuit/queue_length', 100) # number of
    ↪ message points displayed

class pure_pursuit:


    def __init__(self):
        #Topics & Subscriptions,Publishers
        lidarscan_topic = '/scan'
        drive_topic = '/nav'
        map_topic = '/map'
        odom_topic = '/odom'
        path_topic = '/path'
        marker_goal_topic = '/marker_goal'
        trajectory_topic = '/trajectory'

        self.path_poses = None
        self.velocity = VELOCITY
        self.log_counter = 0

        if VISUALIZATION:
            self.init_trail()

        self.path_sub = rospy.Subscriber(path_topic, Path, self.path_callback,
            ↪ queue_size=1)
        self.odom_sub = rospy.Subscriber(odom_topic, Odometry, self.
            ↪ odom_callback, queue_size=1)
        if not BASIC_VELOCITY:
            self.lidar_sub = rospy.Subscriber(lidarscan_topic, LaserScan, self.
                ↪ lidar_callback, queue_size=1) # optional
        self.drive_pub = rospy.Publisher(drive_topic, AckermannDriveStamped,
            ↪ queue_size=1)
        self.marker_goal_pub = rospy.Publisher(marker_goal_topic, Marker,
            ↪ queue_size=1)
        self.trajectory_pub = rospy.Publisher(trajectory_topic, Marker,
            ↪ queue_size=1)

        self.tf_buffer = tf2_ros.Buffer()
        listener = tf2_ros.TransformListener(self.tf_buffer)


    def odom_callback(self, data):
        if self.log_counter % int(LOG_OUTPUT_LENGTH) == 0:
            if LOG_OUTPUT:
                rospy.loginfo('x:␣%s,␣y:␣%s', data.pose.pose.position.x, data.
                    ↪ pose.pose.position.y)
            if VISUALIZATION:
                self.visualize_trail(data)
        self.log_counter += 1

        self.pursuit_algorithm(data)
    def path_callback(self, data):
        self.path_poses = data.poses
        print("Map␣received")

    # Calculate the velocity based on the distance in front of the car
    def lidar_callback(self, data):
        velocity = data.ranges[540] * VELOCITY_GAIN
        if velocity > MAX_SPEED:
            velocity = MAX_SPEED
        if velocity < MIN_SPEED:
            velocity = MIN_SPEED
        self.velocity = velocity

    def pursuit_algorithm(self, data):
        current = data.pose.pose.position
        if self.path_poses is None:
            return
        goal = self.get_goal(current)
        if VISUALIZATION:
            self.visualize_goal(goal)
        local_position = self.get_local_position(goal)
        angle = self.get_steering_angle(local_position)
        self.publish_drive_msg(self.velocity, angle)

    # Return the index and the distance of the nearest point in the path
    def find_nearest(self, current_position):
        # I didn't use math.inf due to compatibility issues
        min_dist = -1
        min_index = 0
        for index, pose in enumerate(self.path_poses):
            path_position = pose.pose.position
            distance = math.dist([current_position.x, current_position.y], [
                ↪ path_position.x, path_position.y])
            if min_dist == -1:
                min_dist = distance
            if distance < min_dist:
                min_dist = distance
                min_index = index
        return min_index, min_dist

    # Returns the position of the goal point in the path
    def get_goal(self, current_position):
        index, distance = self.find_nearest(current_position)
        pose = self.path_poses[index]
        # Avoid infinite loop
        iterations = 0
        while distance < LOOKAHEAD_DISTANCE and iterations <= len(self.
            ↪ path_poses):
            if index < len(self.path_poses) - 1:
                index += 1
            else:
                index = 0
            pose = self.path_poses[index]
            path_position = pose.pose.position
            distance = math.dist([current_position.x, current_position.y], [
                ↪ path_position.x, path_position.y])
            iterations += 1
        return pose

    # Returns a list of 2 elements: x position, y position
    def get_local_position(self, goal):
        local_pose = self.transform_pose(goal, 'map', 'base_link')
        return local_pose.position

    # Get the steering angle from the curvature calculated with the formula
        ↪ GAMMA = 2y/L^2
    def get_steering_angle(self, position):
        y = position.y
        curvature = (2 * y) / (LOOKAHEAD_DISTANCE * LOOKAHEAD_DISTANCE)
        steering_angle = STEERING_GAIN * curvature
        return steering_angle

    def publish_drive_msg(self, velocity, angle):
        drive_msg = AckermannDriveStamped()
        drive_msg.header.stamp = rospy.Time.now()
        drive_msg.header.frame_id = "laser"
        drive_msg.drive.steering_angle = angle
        drive_msg.drive.speed = velocity
        self.drive_pub.publish(drive_msg)

    def transform_pose(self, input, from_frame, to_frame):
        input_pose = input.pose

        pose_stamped = tf2_geometry_msgs.PoseStamped()
        pose_stamped.pose = input_pose
        pose_stamped.header.frame_id = from_frame
        pose_stamped.header.stamp = rospy.Time()

        try:
            # ** It is important to wait for the listener to start listening.
                ↪ Hence the rospy.Duration(1)
            output_pose_stamped = self.tf_buffer.transform(pose_stamped,
                ↪ to_frame, rospy.Duration(1))
            return output_pose_stamped.pose

        except (tf2_ros.LookupException, tf2_ros.ConnectivityException, tf2_ros
            ↪ .ExtrapolationException):
            raise

    def visualize_goal(self, goal):
        # print(goal)
        marker_goal = Marker()
        marker_goal.header.frame_id = 'map'
        marker_goal.header.stamp = rospy.Time.now()
        marker_goal.type = 2
        marker_goal.pose = goal.pose
        marker_goal.scale.x = .3
        marker_goal.scale.y = .3
        marker_goal.scale.z = .3
        marker_goal.color.a = 1
        marker_goal.color.r = 0
        marker_goal.color.g = 0
        marker_goal.color.b = 1
        # marker_goal.color = [1, 0, 1, 0]
        self.marker_goal_pub.publish(marker_goal)

    def init_trail(self):
        self.points = []
        self.trajectory_msg = Marker()
        self.trajectory_msg.header.frame_id = 'map'
        self.trajectory_msg.type = 7
        self.trajectory_msg.scale.x = .3
        self.trajectory_msg.scale.y = .3
        self.trajectory_msg.scale.z = .3
        self.trajectory_msg.color.a = 1
        self.trajectory_msg.color.r = 1
        self.trajectory_msg.color.g = 1
        self.trajectory_msg.color.b = 0
        self.trajectory_msg.pose.position.x = 0
        self.trajectory_msg.pose.position.y = 0
        self.trajectory_msg.pose.position.z = 0
        self.trajectory_msg.pose.orientation.w = 0
        self.trajectory_msg.pose.orientation.x = 0
        self.trajectory_msg.pose.orientation.y = 0
        self.trajectory_msg.pose.orientation.z = 0

    def visualize_trail(self, data):
        self.trajectory_msg.header.stamp = rospy.Time.now()
        point = Point(data.pose.pose.position.x, data.pose.pose.position.y,
           ↪ data.pose.pose.position.z)
        self.points.append(point)
        if (len(self.points) > QUEUE_LENGTH):
            self.points.pop(0)
        self.trajectory_msg.points = self.points
        self.trajectory_pub.publish(self.trajectory_msg)

def main(args):
    rospy.init_node("pure_pursuit_node", anonymous=True)
    rfgs = pure_pursuit()
    rospy.sleep(0.1)
    rospy.spin()

if __name__ == '__main__':
    main(sys.argv)
```