TECHNISCHE UNIVERSITÄT WIEN

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

Advanced Racing 1

2022-05-31

Student data

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

Transparency of contribution

- **Danilo Castiglia**: Algorithms tuning and comparisons, hardware car planning and pure pursuit attempts
- **Artur Chaves**: Disparity extender tune
- **Severin Jäger**: Path planning improvements
- **Johannes Windischbauer**: Timer node and hardware struggles

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.

TECHNISCHE UNIVERSITÄT WIEN

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

Advanced Racing 1

2022-05-31

Team data

Team number: 2

1 Goals

Based on our abstract submitted in lab 6, we derived the following goals:

1. Create a lap timer node for objective comparison of our algorithms
2. Improve the path planner implementation w.r.t. to lap time by not only considering the shortest path but also aiming for smoother trajectories.
3. Tune both disparity extender and pure pursuit for fast lap times in the simulator.
4. Assess the feasibility of planning-based driving on hardware.

2 Implementation

2.1 Lap Timer

To be able to quantify our tuning approaches we implemented a very simple timer node in python. The timer node is tune-able by setting a left and right position for the start. Once the node is started the timer starts running, so the first lap should always be discarded. If the car then passes the x-axis a variable is set to true and after passing the finish line the time will be printed and the variable set to false. The approach is quite simple but works reliable on all tested tracks. The code for the timer node can be found in listing 1.

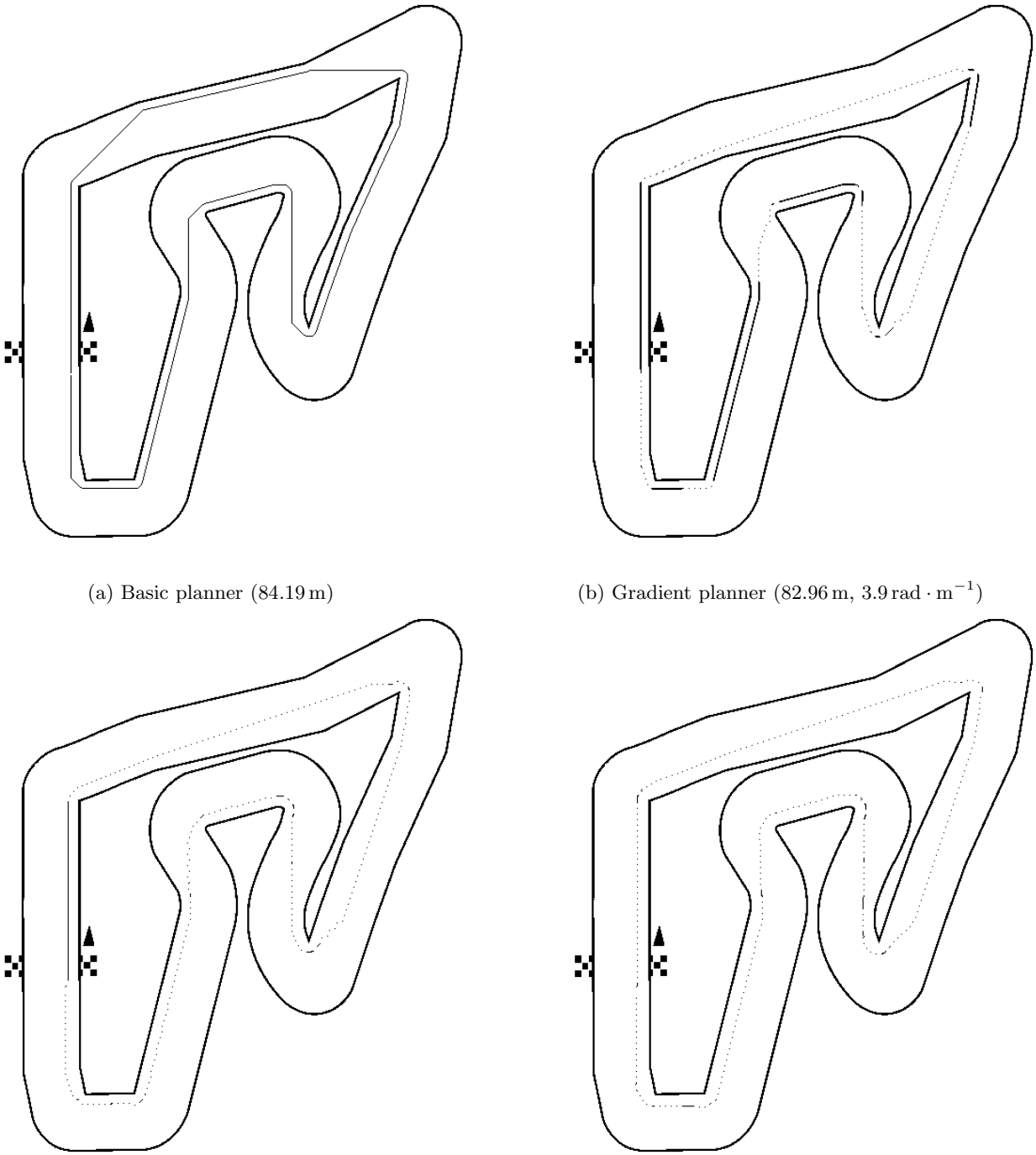
2.2 Path Planning

The gradient-based planner we implemented in lab 6 already outperformed the A*-like reference algorithm in terms of path length. We also did first steps towards smoothing the generated paths by considering not only the current but also the previous and the next gradient. Still, this hardly considers the velocity at which the path can be followed safely. Our initial idea was superposing some sort of curve radius field to the distance field and take the gradient of the sum. However, calculating the curve radius has to be redone after every step and requires some lookahead horizon. This leads to high complexities and run times (as values have to be computed for every pixel). Instead, we decided to compute an easier to drive path by considering not only the distance towards the finish line but also the distance to the wall. The underlying notion is that the car only drives close to a wall if this leads to a significantly shorter distance.

Formally, this means that the path follows the gradient of the following cost function

$$J(\mathbf{r}) = (1-\alpha)d_{finish}(\mathbf{r}) + \frac{\alpha}{d_{wall}(\mathbf{r})^2 + \beta}$$

with $\mathbf{r} = [x, y]$. The parameter α represents the tradeoff between wall distance and short path. To avoid divisions by zero, a small offset β is used.



(a) Basic planner (84.19 m)

(b) Gradient planner (82.96 m, 3.9 rad · m^{-1})

(c) Improved gradient planner considering wall clearance (84.01 m, 5.36 rad · m^{-1})

(d) Improved gradient planner considering wall clearance with smoothing (84.22 m, 3.02 rad · m^{-1})

Figure 1: Path planner algorithms on the f1_aut_wide map

Figure 1 compares the paths calculated by different variants of our algorithm. Clearly, the gradient-based planner reduces the path length. Once clearance is considered, the path is less risky as the wall distance is reduced where it is not too much of a detour. Finally, the smoothing reduces the steering effort (which is the sum of all angular changes between waypoints divided by the waypoint distances). This gains are also directly translating to lap speeds. Whereas (for the same disparity extender and speed controller) the basic planner yields a lap time of

19.24 s, a smoothed gradient planner achieves 18.42 s. Finally, the planner considering the wall clearance decreases the lap time to 18.04 s.

One other aspect worth mentioning is that an aggressively tuned pure pursuit smoothens the path due to the lookahead distance quite significantly, s. Figure 2. This makes path interpolation or similar techniques in the planning phase quite redundant from our point of view.

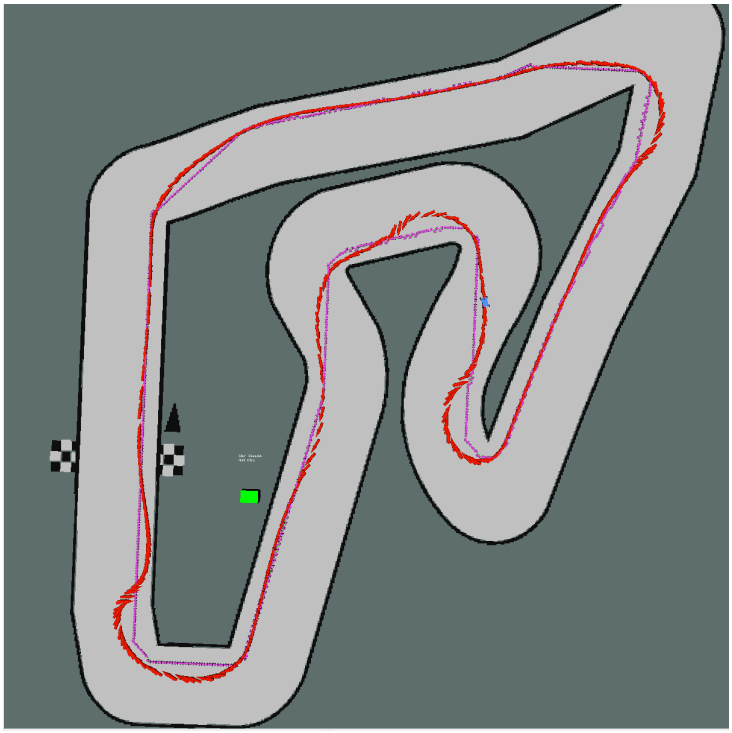


Figure 2: Path smoothing by pure pursuit. The planned path is drawn in blue, the actual trajectory in red.

Furthermore, we tried our path planner with maps created by Google cartographer on the hardware. To achieve good planning and to reduce localisation problems, we manually corrected the maps. Mainly, we filled holes in both the track and the walls. In addition, the x and y axis of the images are interchanged in comparison the the maps available in the simulator. Once this is taken into account, the planner works on SLAM maps reasonably. An example is given in Figure 3.

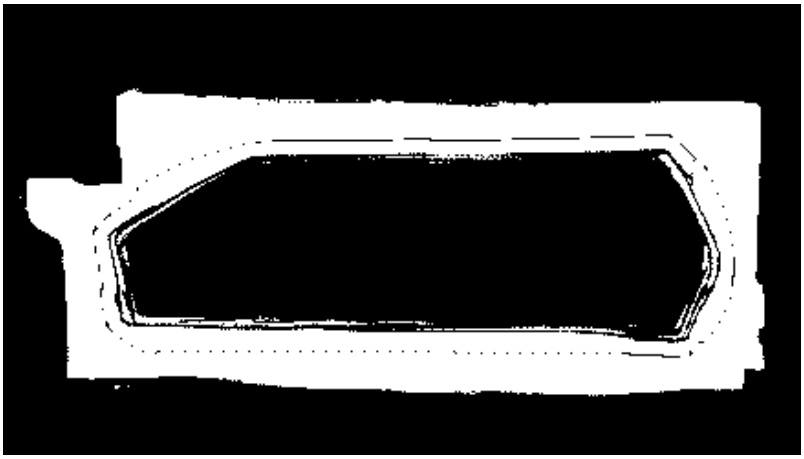


Figure 3: Planned path on a corrected Informatikhörsaal map

Map	Disparity Extender	Pure Pursuit
f1_aut_wide	17.75s	18.04s
f1_gbr	43.30s	43.59s
f1_esp	47.71s	48.92s
Informatikhörsaal (SLAM)	11.32s	9.81s

Table 1: Comparison of the timing results of different algorithms in different maps on the simulator.

2.3 Velocity calculation

For this lab we tried to use the same speed function in order to achieve comparable results. We used a linear function based on the distance in front of the car, limited by a maximum and a minimum speed. In addition to that we have the velocity gain that is the coefficient of the function. This 3 parameters are used to control the speed of the car. We also have a steering gain that is similar to the velocity gain, but is used to control how big should the steering angle be.

We tuned all these parameters for both the algorithms and used the timer node to measure which one of the algorithms performs better. We run these experiments in the simulator and reported all the results in the Table 1.

Disparity Extender performed better in 3 over 4 maps. The problem of Pure Pursuit is that using the velocity defined above the car drifts heavily. The reason can be in the nature of Pure Pursuit that has sharp turns, we tried to limit this by reducing the steering gain but then the car wasn't able anymore to complete the circuit without crashing. Another problem is the overshooting, as we can see in Figure 2, because the car, trying to follow the path, faces the wall and then slows down (using the velocity defined above).

3 Hardware

One very important part of our lab 7 goals was to try our algorithms on hardware. Unfortunately, we had quite some trouble to get everything to run in the limited hands-on time we had in this lab. The first hours were used to try to run the path planner on the car which turned out to be impossible due to version and dependency issues (python2 on the hardware and python3 in the simulator). Ultimately, we decided that it is not necessary to run the path planning on the car and we are instead running it on our local machines. To be able to do this the implementation of the path planner was slightly changed to record a rosbag instead of publishing the path directly so that we can replay the bag on the hardware care.

Once we figured this out we had trouble with the path planner implementation on our newly slammed maps. The newly slammed maps had to be pre-processed manually to fill all gaps. After quite some debugging we were however able to run the path planning and try pure pursuit on the car which then turned out to be unbelievably unusable due to performance issues with amcl.

4 Further Topics

Disparity Extender Improvements with Pre-Calculated Paths: As discussed in the lecture we are going to try to include knowledge we are getting from pre-calculating the path for tuning the disparity extender algorithm so that we can brake harder and avoid possible follow-the-gap traps.

Dynamic speed calculation: We are continuing to improve our speed calculation, possibly also by utilizing the path.

As we are mainly aiming at a good performance in the final race, we focus on algorithms that work on hardware well and which can be tuned quickly in a race setting. As we consider our current In general we are going to focus on reactive driving since our car doesn't provide IMU data and the AMCL does not work properly. If there is a chance to swap this with the MIT particle filter we might be able to improve our algorithms even further and thus switch to a more sophisticated planner. However, we will assess in the simulator first if the planning-based method is able to outperform our disparity extender. We also want to spend more time trying stuff on the hardware since the hardware to reality gap seems quite big with some measures.

Appendix

Listing 1: Python implementation of the timer

```
#!/usr/bin/env python3
from __future__ import print_function
from curses import raw
from platform import java_ver
from re import T
import sys
import math
import time

#ROS Imports
import rospy
from nav_msgs.msg import Odometry

# TIMER PARAMS
START_X1 = rospy.get_param('/timer/x1', 0)
START_Y1 = rospy.get_param('/timer/y1', -2)
START_X2 = rospy.get_param('/timer/x2', 0)
START_Y2 = rospy.get_param('/timer/y2', 2)

class Timer:
    def __init__(self):
        self.start = time.time()
        self.below_zero = False
        self.lap_counter = 1
        odom_topic = '/odom'
        self.odom_sub = rospy.Subscriber(odom_topic, Odometry, self.
            ↪ odom_callback, queue_size=1)

    def odom_callback(self, data):
        if data.pose.pose.position.x < 0 and not self.below_zero:
            self.below_zero = True
        if data.pose.pose.position.x > 0 and self.below_zero:
            if data.pose.pose.position.y > START_Y1 and data.pose.pose.position
                ↪ .y < START_Y2:
                lap = time.time()
                rospy.loginfo("Lap␣%i:␣%f", self.lap_counter, lap - self.start)
                self.start = lap
                self.below_zero = False
                self.lap_counter += 1


def main(args):
    rospy.init_node("timer_node", anonymous=True)
    timer = Timer()

    rospy.sleep(0.1)
    rospy.spin()

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

Listing 2: Python implementation of the planner

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

#ROS Imports
import rospy
import rosbag
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__

MAX_FLOAT = 99999.9  # avoid problems with python2
DEBUG = False

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

        # Parameters
        self.IS_SLAM_MAP = rospy.get_param('/planner/is_slam_map', "False")  #
            ↪ image created with cartographer?
        self.MAP_NAME = rospy.get_param('/planner/map_name', "")  # name for
            ↪ the exported map
        self.BAG_NAME = rospy.get_param('/planner/bag_name', "")  # name for
            ↪ the exported rosbag

        # 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)

        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.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.CLEARANCE_WEIGHT = rospy.get_param('/planner/clearance_weight',
            ↪ 0.33)
        self.MAX_REASONABLE_ANGLE = math.radians(60.0)
        self.MIN_LAP_LENGTH = 20.0

        # 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)

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

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

    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.resolution = data.info.resolution
        if self.IS_SLAM_MAP:
            self.shape = (data.info.height, data.info.width)
            self.start_position = (data.info.origin.position.y,
                                   data.info.origin.position.x)
        else:
            self.shape = (data.info.width, data.info.height)
            self.start_position = (data.info.origin.position.x,
                                   data.info.origin.position.y)

        self.start_pixel = (int(-self.start_position[0]/self.resolution),
                            int(-self.start_position[1]/self.resolution))

        print(self.start_pixel)

        # 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(map)
        if DEBUG:
            self.save_map(driveable_area, "1_drivable_area")
```

```
driveable_area = self.add_safety_foam(driveable_area)
if DEBUG:
    self.save_map(driveable_area, "2_drivable_area_safety")

clearances = self.get_clearances(driveable_area)
if DEBUG:
    self.save_map(clearances/np.max(clearances), "3_clearances")

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,
       ↪ clearances, 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 and lap_length > self.MIN_LAP_LENGTH:
        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
if DEBUG:
    self.save_map(distances_print/np.max(distances_print), "4_distances
       ↪ ")

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

try:
    bag = rosbag.Bag(self.BAG_NAME + '.bag', 'w')
    bag.write(self.path_topic, self.path_msg)
    rospy.loginfo("Bag␣file␣written␣to␣" + self.BAG_NAME + '.bag')
finally:
    bag.close()
```

```
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 get_clearances(self, driveable_area):
    OFFSET = 0.01  # prevents division by 0
    clearances = np.zeros(self.shape, dtype=float)

    indices = np.argwhere(driveable_area)

    # find track edge
    binary_image = morphology.binary_erosion(driveable_area)
    indices_out = np.argwhere(driveable_area!=binary_image)

    for index in indices:
        clearances[index[0], index[1]] = np.min(np.linalg.norm(indices_out
           ↪ - index, axis=1))

    clearances = clearances**2 + OFFSET
    clearances = 1/clearances

    return clearances

def preprocess_map(self, data):
    if self.IS_SLAM_MAP:
```

```
            map_data = np.asarray(data.data).reshape((data.info.height, data.
                ↪ info.width)) # parse map data into 2D numpy array
        else:
            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 < 0.65  # make binary occupancy map
        return map_binary

    def get_driveable_area(self, 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, clearances, 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()

steering_effort = 0.0

best_x = -1
best_y = -1

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

# TODO steering effort

if best_x == -1 and best_y == -1:
    rospy.logerr("No␣valid␣path␣found␣from␣this␣starting␣position.␣Make
       ↪ ␣sure␣the␣SAFETY_DISTANCE␣does␣not␣block␣the␣track␣and␣
       ↪ consider␣increasing␣START_OFFSET.")
    exit(-1)

last_x = x
last_y = y
x = best_x
y = best_y
last_waypoint = np.array([x, y])
last_wp_orientation = np.arctan2(last_x-x, last_y-y) + np.pi

# 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
        field = (1-self.CLEARANCE_WEIGHT)*distances + self.CLEARANCE_WEIGHT
            ↪  * clearances
        g_x, g_y = np.gradient(field)
        orientation = np.arctan2(g_x, g_y)+np.pi

        # self.save_map((orientation+np.pi)/(np.max(orientation)* 2 * np.pi
            ↪ ), "6_orientation")
        dir = orientation[x, y]

        i = 0
        while(distance > self.STEP_SIZE-1.0):
            path[x,y] = 0.0
            path_point = np.array([x, y])

            if np.linalg.norm(path_point - last_waypoint) >= self.
                ↪ WAYPOINT_DISTANCE:
                 wp_orientation = np.arctan2(last_waypoint[0]-path_point[0],
                     ↪  last_waypoint[1]-path_point[1]) + np.pi
                 self.add_pose_to_path(path_msg, x, y, orientation=
                     ↪ wp_orientation)
                 path_length += np.linalg.norm(path_point - last_waypoint)

                 ang = np.abs(wp_orientation-last_wp_orientation)
                 if ang > np.pi:
                     ang = ang - 2*np.pi
                 steering_effort += (ang/np.linalg.norm(path_point -
                     ↪ last_waypoint))

                 last_waypoint = path_point
                 last_wp_orientation = wp_orientation


            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

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

                    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
                else:
                    dir = orientation[x, y]
                    next_step = np.sqrt(2)

                last_x = x
                last_y = y
                x = int(x + next_step * np.sin(dir)+0.5)
                y = int(y + next_step * np.cos(dir)+0.5)
                distance = distances[x, y]

                i = i+1
                if i > 50000:
                    rospy.logerr("No␣path␣terminating␣at␣the␣start␣found.␣
                       ↪ Consider␣reducing␣CLEARANCE_WEIGHT.")
                    exit(-1)

            # 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)

            rospy.loginfo("Steering:␣" + str(steering_effort))
        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()
        if self.IS_SLAM_MAP:
            pose_msg.pose.position.x = self.resolution*y + self.start_position
               ↪ [1]
```

```
            pose_msg.pose.position.y = self.resolution*x + self.start_position
               ↪ [0]
        else:
            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 3: Planner launch file

```
<?xml version="1.0"?>

<!-- Run the planner on a PC to avoid installation problems on the car -->
<!-- The calculated path is written to a bag file which is read by pure pursuit
   ↪  -->

<launch>
    <!-- bunch_of_sticks / f1_aut_wide / f1_aut_wide_obstacles / f1_aut /
       ↪ f1_mco / f1_gbr / f1_esp / test / infhs_slam_2-->
    <arg name="map"  default="f1_aut_wide"/>
    <arg name="is_slam_map"  default="False"/>
    <arg name="bag_name"  default="$(find␣pure_pursuit)/maps/$(arg␣map)_path"/>

    <!-- Map server-->
    <node pkg="map_server" name="map_server" type="map_server" args="$(find␣
       ↪ f1tenth_simulator)/maps/$(arg␣map).yaml"/>
```

```
    <arg name="wall_distance"  default="0.9"/>
    <arg name="waypoint_distance"  default="0.25"/>
    <arg name="start_line_step"  default="4"/>
    <arg name="use_gradient_descent"  default="True"/>
    <arg name="gradient_step_size"  default="6"/>
    <arg name="gradient_weight_past"  default="0.35"/>
    <arg name="gradient_weight_future"  default="0.15"/>
    <arg name="clearance_weight"  default="0.9"/>

    <node name="planner" pkg="pure_pursuit" type="planner.py" output="screen">
         <param name="wall_distance"  value="$(arg␣wall_distance)"/>
         <param name="waypoint_distance"  value="$(arg␣waypoint_distance)"/>
         <param name="start_line_step"  value="$(arg␣start_line_step)"/>
         <param name="map_name"  value="$(arg␣map)"/>
         <param name="bag_name"  value="$(arg␣bag_name)"/>
         <param name="use_gradient_descent"  value="$(arg␣use_gradient_descent)
            ↪ "/>
         <param name="gradient_step_size"  value="$(arg␣gradient_step_size)"/>
         <param name="gradient_weight_past"  value="$(arg␣gradient_weight_past)
            ↪ "/>
         <param name="gradient_weight_future"  value="$(arg␣
            ↪ gradient_weight_future)"/>
         <param name="is_slam_map"  value="$(arg␣is_slam_map)"/>
         <param name="clearance_weight"  value="$(arg␣clearance_weight)"/>
    </node>
</launch>
```

Listing 4: Python implementation of the pure pursuit node

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

#ROS Imports
import rospy
import rosbag
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

# 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.path_pub = rospy.Publisher(path_topic, Path, queue_size=10, latch=
            ↪ True)

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

        bag_path = rospy.get_param('/pure_pursuit/bag_name', "") + ".bag"
```

```
        bag = rosbag.Bag(bag_path)
        rospy.loginfo("Read␣bag␣file␣from␣" + bag_path)

        for topic, msg, t in bag.read_messages(topics=[path_topic]):
            self.path_pub.publish(msg)

    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[int(len(data.ranges) / 2)] * 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)
```

Listing 5: Pure pursuit launch file

```
<?xml version="1.0"?>

<launch>
    <!-- bunch_of_sticks / f1_aut_wide / f1_aut_wide_obstacles / f1_aut /
       ↪ f1_mco / f1_gbr / f1_esp / test / infhs_slam_2-->
    <arg name="map"  default="f1_aut_wide"/>
    <arg name="bag_name"  default="$(find␣pure_pursuit)/maps/$(arg␣map)_path"/>

    <include file="$(find␣f1tenth_simulator)/launch/simulator.launch">
        <arg name="map" value="$(find␣f1tenth_simulator)/maps/$(arg␣map).yaml"/
           ↪ >
        <arg name="map_frame"  value="map"/>
    </include>

    <arg name="lookahead_distance" default="2.0"/>
    <arg name="steering_gain" default="0.63"/>
    <arg name="max_speed"  default="6"/>
    <arg name="min_speed"  default="2.5"/>
    <arg name="velocity_gain"  default="1.65"/>
    <arg name="basic_velocity"  default="False"/>
    <!-- Visualization parameters -->
    <arg name="visualization" default="False"/>
    <arg name="log_output_length" default="100"/>
    <arg name="log_output" default="False"/>
    <arg name="queue_length" default="100"/>
    <node name="pure_pursuit" pkg="pure_pursuit" type="pure_pursuit.py" output=
       ↪ "screen">
        <param name="lookahead_distance" value="$(arg␣lookahead_distance)"/>
        <param name="steering_gain" value="$(arg␣steering_gain)"/>
        <param name="max_speed"  value="$(arg␣max_speed)"/>
        <param name="min_speed"  value="$(arg␣min_speed)"/>
        <param name="velocity_gain"  value="$(arg␣velocity_gain)"/>
        <param name="basic_velocity"  value="$(arg␣basic_velocity)"/>
        <param name="visualization" value="$(arg␣visualization)"/>
        <param name="queue_length" value="$(arg␣queue_length)"/>
        <param name="log_output_length" value="$(arg␣log_output_length)"/>
        <param name="log_output" value="$(arg␣log_output)"/>
        <param name="bag_name" value="$(arg␣bag_name)"/>
    </node>

    <include file="$(find␣timer)/launch/timer.launch">
    </include>
</launch>
```