MULTI-AGENT NON-HOLONOMIC RACING: HARDWARE, SOFTWARE, AND ALGORITHMS

NOVEMBER 21ST, 2022

Abstract

This document provides an overview of the hardware and software components required for building and operating a Multi-agent System for non-Holonomic Racing (MuSHR). It uses the MuSHR developed by the University of Washington as an example vehicle, but the concepts and methods are general and applicable to other platforms. The document covers the topics of perception, motion planning, and motion control, and discusses the various options and trade-offs for each module. It also provides some general resources for further learning and development.

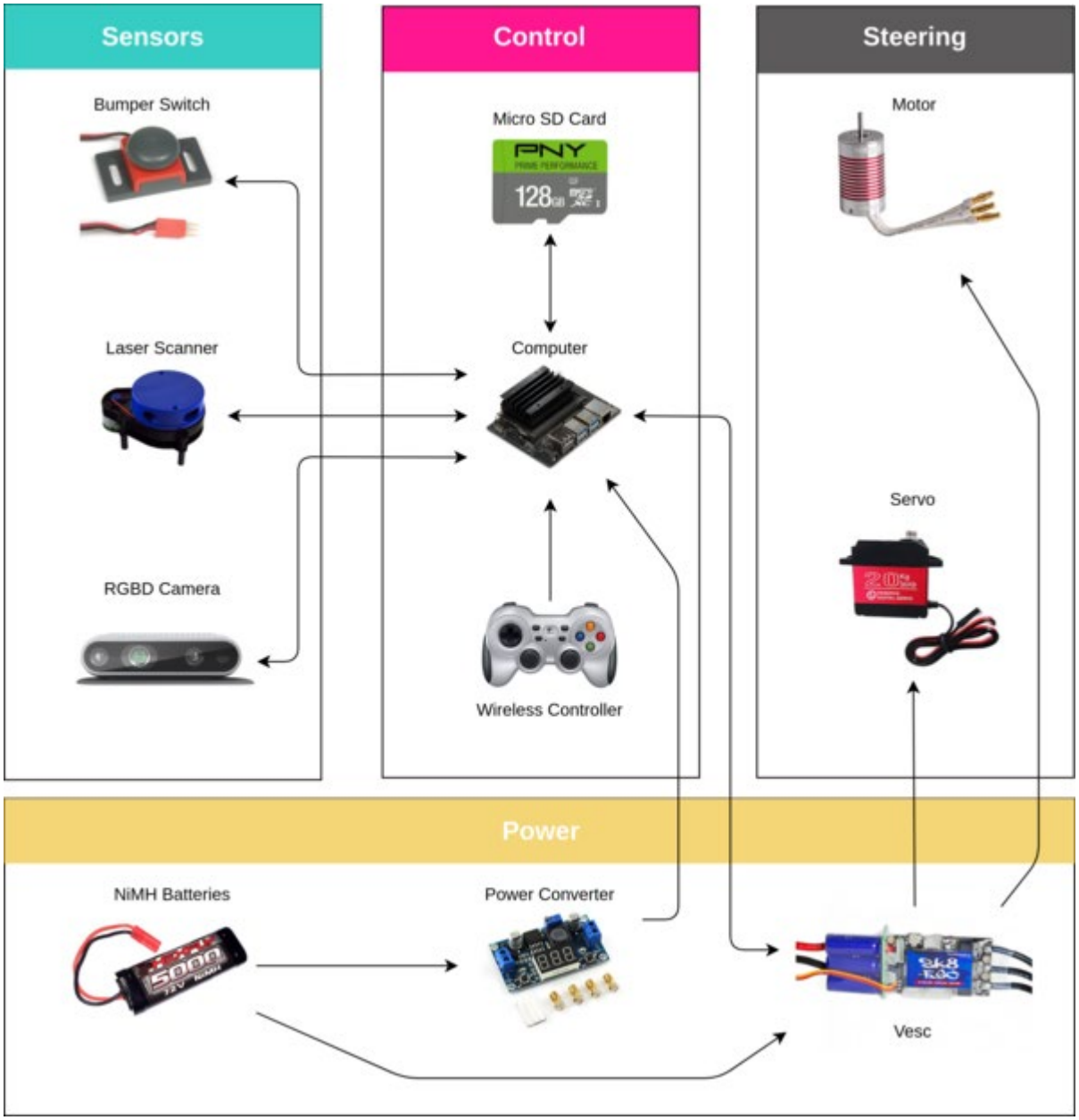
Tunisian Space Association
الجمعية التونسية للفضاء

Elyes Khechine
elyeskhechine@gmail.com
Space Robotics Team

GitHub

Contents

I. Hardware Overview: Multi-agent System for non-Holonomic Racing (MuSHR)



- **Chassis (Redcat Racing Blackout SC 1/10) :** The car chasis. It has adjustable 4x4 suspension and non-flat tires. It is also used for mounting things to it, including the housing which contains the computer and sensors.
- **Computer (Jetson Nano):** This the computer that runs the software on the car. You can connect to the computer in 3 ways primarily: ssh through local network to static IP, connecting to car's network and using ssh, or plugging a monitor, keyboard, and mouse into the computer directly.
- **Motor (Jrelecs F540 3930KV):** The single DC motor that powers all four wheels. The motor makes the car move and is controlled by the VESC.

- **Servo (ZOSKAY 1X DS3218):** A servo is another type of motor but is better at going to specific angles along its rotation than rotating continuously. The servo's job is to steer the front wheels by taking in steering angle commands. The servo is also controlled by the VESC.
- **VESC (Turnigy SK8-ESC):** The VESC is responsible for taking high level control commands like steering angle and velocity and converting that to power/angle commands for the motor/servo. The VESC has an associated ROS node. This node takes your ROS ackermann_msgs/AckermannDriveStamped message and converts that to VescStateStamped (power), and Float64 (steering angle) messages. These messages are something the physical VESC can use to control the motor and servo.
- **NiMH Battery (Redcat Racing HX-5000MH-B):** There are 2 batteries. One to power the motor and in turn the VESC (because the motor power flows through the VESC). And a second to power the computer and sensors. Many of the issues (particularly with the VESC) can stem from one of these batteries having a low battery, so it is good to check them regularly.
- **Power Converter (DZS Elec LM2596):** This power converter converts the higher voltage of the battery to the necessary 5V max for the computer.
- **RGBD Camera (Realsense D435i):** This Intel Realsense camera can publish both rgb and depth. It has a associated node so you can just subscribe to the topic to use the images! OpenCV even has a function for converting ROS Image messages to something OpenCV can use. Note depth cameras are different from stereo cameras (how humans do it). They both provide the same output, but depth cameras project a IR pattern and use a IR camera to see the deformation of that pattern to compute depth. Depth cameras will also sometimes use stereo in addition to make their measurements more accurate. It publishes to the /camera topics. This camera can also publish IMU measurments, but they currently are not being used.
- **Laser Scanner (YDLIDAR X4):** This 360 degree sensor works by having a 1D laser range finder spin around. With each distance reading there is an associated angle the range finder was at. It publishes an array of distance and angle measurements to the /scan topic.
- **Wireless Controller (Logitech F710):** This provides controls to the cars! It doesn't just have to be use for teleop it can be used by your programs for most anything. It publishes on the /joy topic.
- **Bumber Switch (Vex Bumper Switch):** This is a button on the front of the car that you can use to indicate a collision. It publishes a binary signal to **/push_button_state**.
- **Micro SD Card:** Storage for the OS and any logs. What's great about this is if you want to switch cars you can simply switch SD cards.

II. Software Overview

A. General Architecture

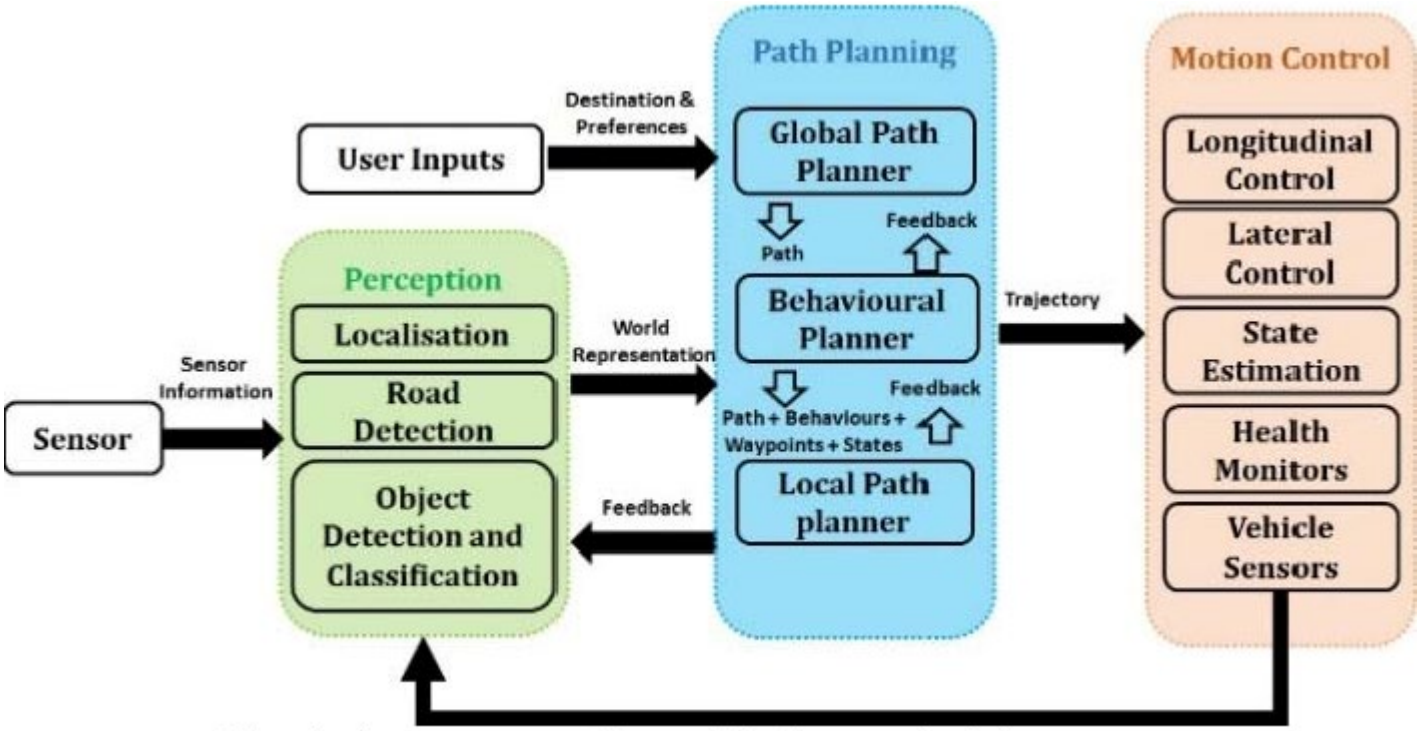
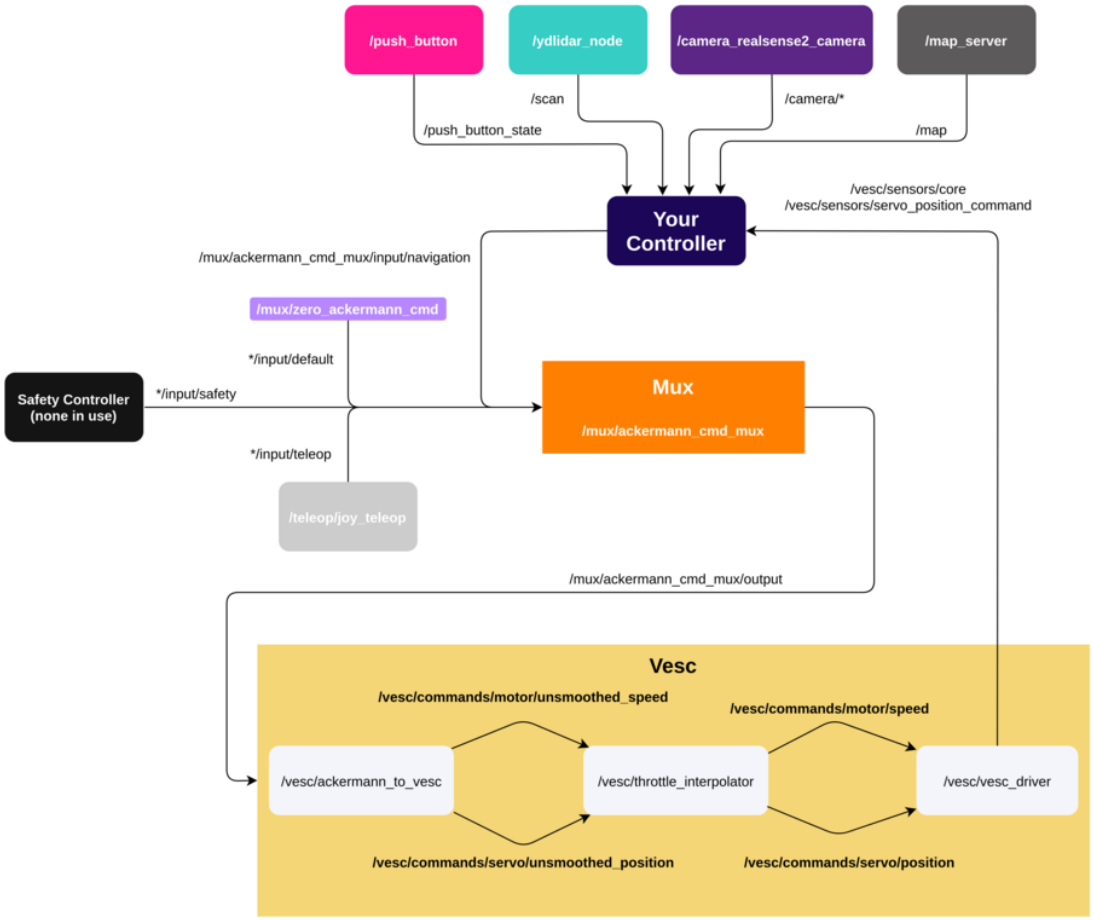


Fig. 1. Autonomous Control Software Architecture.

B. MuSHR Software Architecture



- The sensor nodes take the raw input, does some processing, and convert it to ROS friendly topics:

- The map server converts a .yaml map file to a ros topic.

- Your controller (orange) takes in sensor topics coming in from the each sensors' ROS node and the map. It then outputs some command to drive the car.

- That command is put into the mux which listens on multiple /mux/ackermann_cmd_mux/input channels and selects the highest priority.

*/input/default is a zero throttle/steering command that is passed whenever your controller and teleop are not publishing.

- Currently, MuSHR does not have an explicit safety controller publishing to the */input/safety topic. The mux priorities can be found mushr_base/ackermann_cmd_mux/param/mux.yaml and they are listed in order of priority below.

1. Safety

2. Teleop

3. Navigation

4. Default

Once the highest priority command is output it goes to the VESC. The VESC smoothes the command by clipping the min/max of the steering/throttle so we don't try to turn the wheels 180 degrees for example. It then provides that to the VESC driver which directly controls the motors.

C. MuSHR Navigation Stack

1. Components

At the highest level MuSHR's navigation stack consists of three principal components:

- Receding Horizon Controller (RHC) Node
- Localization Node
- Planner Node

2. Running the navigation stack

To operate the navigation stack, we will use foxglove to send pose targets to the vehicle. When operating in the real world, the pose estimate of the car may be incorrect. You can correct this by providing the particle filter with the correct pose estimate using the Set Pose Estimate button on the bottom right of the foxglove window and then using the button to publish clicked points.

Note: When publishing a pose, the pose will correspond to the pose at the tip of the arrow and not the base of the arrow.

DEMO: https://mushr.io/tutorials/autonomous-navigation/final_vid_stack.mp4

III. Perception

A. Localization

In order for the controller to know where it is, and therefore also whether it is in the proximity of known obstacles in the map, it must know its location. Solving this problem is called "localization".

1. Options

i. AI-IMU Dead-reckoning:

https://arxiv.org/pdf/1904.06064.pdf

ii. Using Beacons and Kalman Filter Technique:

Mobile Robot Localization Using Beacons and the Kalman Filter Technique for the Eurobot Competition: https://www.researchgate.net/publication/226622829_Mobile_Robot_Localization_Using_Beacons_and_the_Kalman_Filter_Technique_for_the_Eurobot_Competition

file:///C:/Users/Elyes/Downloads/paperVisualLocalization.pdf

iii. An Accurate Dead Reckoning Method based on Geometry Principles for Mobile Robot Localization:

An Accurate Dead Reckoning Method based on Geometry Principles for Mobile Robot Localization: https://www.researchgate.net/publication/269669077_An_Accurate_Dead_Reckoning_Method_based_on_Geometry_Principles_for_Mobile_Robot_Localization

B. Visual Perception

1. Map Feature Detection: Case of ARuco Markers

https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html

Other resourcres:

https://docs.google.com/spreadsheets/d/1fCicpogaaT-x8NhKWfnyODmS9t_mLWDoEAyf7hP_Z8o/edit#gid=0

2. Object Detection

Presentation: https://www.canva.com/design/DAE9f-Tgvvs/oexk4o3KsXyn9wFWc5nuZw/edit?utm_content=DAE9f-Tgvvs&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

YOLOR research paper: https://arxiv.org/pdf/2105.04206.pdf

YOLOR Source code: https://github.com/WongKinYiu/yolor

YOLOR Projects: https://store.augmentedstartups.com/

YOLOR implementation tutorial: https://www.youtube.com/watch?v=AxHJaFF7eKs&ab_channel=AugmentedStartups

YOLOR tutorials: https://www.youtube.com/playlist?list=PL_NjiOJOuXg3T0qOj-quorbrcqrnkahqm&ab_channel=AugmentedStartups

YOLOR explanation : https://www.youtube.com/watch?v=vjjPrfmxXdA

GitHub

3. MuSHR Multi-agent Car Detection and Yolo Learning

=> Train a model to detect MuSHR cars.

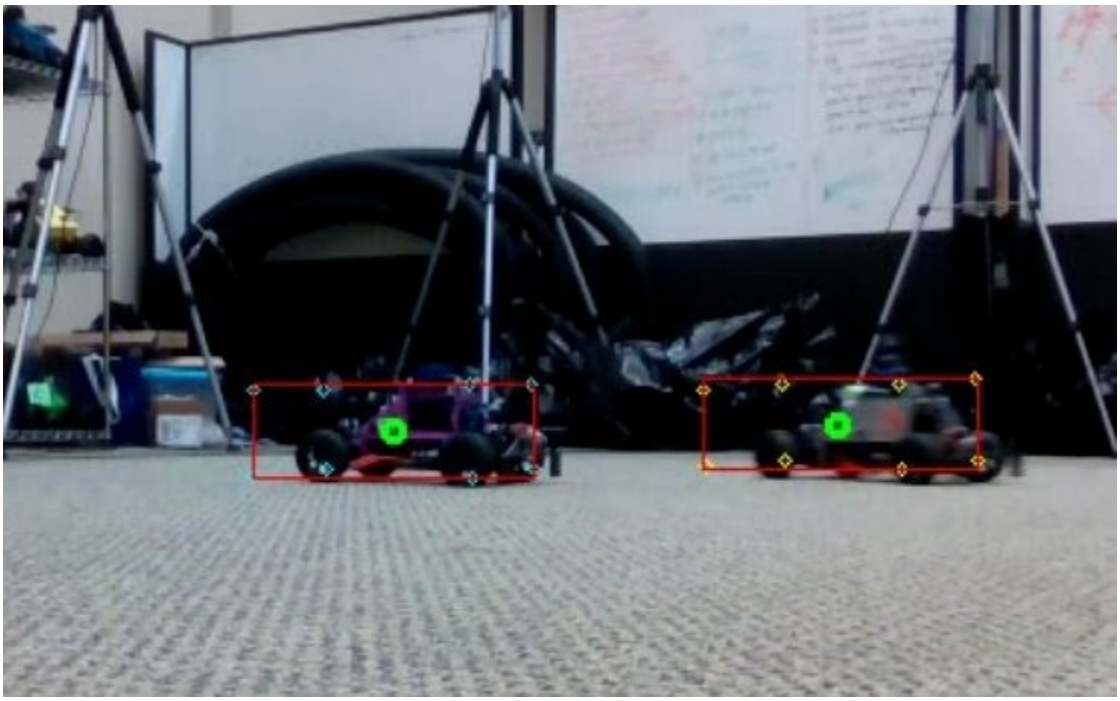
In a multi-agent system, car detection is one of the fundamental processes that must take place in order for cars to make decisions from their observations. This applies to collision avoidance & navigation, as well as any other form of cooperative task. This tutorial will provide instructions for producing bounding prisms to label cars based on mocap car pose data, and training a model from your newly labeled data.

- Recommended Readings:

- Projecting a TF frame onto an image (C++)
- Image Projection

i. Mathematical Explanations

It is important to remember that the sensor measurements from the cars and mocap all have their own frame of reference. In order to combine these into a bounding box within a single car's camera frame, we need to be able to switch between frames of reference in a computationally effective way. This is done through the use of transforms. There are 4 important frames of reference: the world (where the mocap marker is tracked), the base_link (root of a car's tf tree), the camera on the car, and the centroid of a car (used to generate a bounding prism). Being able to move between these frames of reference is necessary to derive the true position of a car on a camera frame, and create a corresponding label.



In this example, car 1 is being viewed by car 2's camera. Let x_T_y mean the transform of x w.r.t. y:

- marker1_T_cam2 = base2_T_cam2 * marker2_T_base2 * world_T_marker2 * marker1_T_world
- center1_T_cam2 = marker1_T_cam2 * base1_T_marker1 * center1_T_base1

marker1_T_world and marker2_T_world are given by the mocap data world_T_marker2 is the inverse of marker2_T_world base_T_cam, marker_T_base, center_T_base are all constants depending on how you set up your system

ii. Bag Creation:

Whether you create your own dataset using your own bag(s), or you use ours, these are the key ROS topics for each car:

- d435 color camera: /carXX/camera/color/image_throttled
- mocap pose: /vrpn_client_node/carXX/pose
- camera parameters: /carXX/camera/color/camera_info

iii. Creating Your Dataset:

Now that you have a bag, you will use gen_darknet_label.py to create your labels & dataset in the darknet format. The implementation of YOLOv5 we are using requires the dataset to be in darknet format:

- bounding boxes are represented by (x_center, y_center, w, h)
- each value is in terms of the dimension of the image between 0 and 1

Create a dataset directory, and create two subdirectories: images & labels. Run the gen_darknet_label.py script to populate the directories with images and labels. When finished, your dataset directory should look like this:

```
dataset/

      images/
            img_000000.jpg
            img_000001.jpg
            ...
      labels/
            img_000000.txt
            img_000001.jpg
            ...
```

[GitHub: pose detection and label generation]

iv. Labeling Implementation

The first thing the script will do is process the entire bag looking for the topics listed above. You will want:

- The pose of each car
- The camera frames and camera info from one car

```

import rosbag
import numpy as np
```

```
  import tf
  import rospy
  import cv2
  from cv_bridge import CvBridge
  from image_geometry import PinholeCameraModel

  PATH_TO_INPUT_BAG = '/home/tudorf/mushr/catkin_ws/src/learning-image-geometry/car37_longtrim.bag'
  OUTPUT_VIDEO_NAME = 'test'
  OUTPUT_VIDEO_FPS = 9

  transformerROS = tf.TransformerROS()
  bridge = CvBridge()
  camModel = PinholeCameraModel()
  vid_out = cv2.VideoWriter(OUTPUT_VIDEO_NAME + '.avi', cv2.VideoWriter_fourcc(*'MJPG'), OUTPUT_VIDEO_FPS, (640,480))

  def translate_transform(p):
        t = np.eye(4)
        t[0,3] = p[0]
        t[1,3] = p[1]
        t[2,3] = p[2]
        return t

  def inverse_transform(t):
        R_inv = t[:3,:3].T
        p_inv = np.matmul(-R_inv,t[:3,3])
        t_inv = np.eye(4)
        t_inv[0,0] = R_inv[0,0]
        t_inv[0,1] = R_inv[0,1]
        t_inv[0,2] = R_inv[0,2]
        t_inv[1,0] = R_inv[1,0]
        t_inv[1,1] = R_inv[1,1]
        t_inv[1,2] = R_inv[1,2]
        t_inv[2,0] = R_inv[2,0]
        t_inv[2,1] = R_inv[2,1]
        t_inv[2,2] = R_inv[2,2]
        t_inv[0,3] = p_inv[0]
        t_inv[1,3] = p_inv[1]
        t_inv[2,3] = p_inv[2]
        return t_inv

  def get_corners(center_T_cam):
        # center_T_cam is the transform of centerXX in terms of camYY, where we want the corners of carXX in terms of camYY
        # dx,y,z are the dimensions of the car in the x,y,z directions from the center
        dx = 0.22
```

```
        dy = 0.134
        dz = 0.079
        return [
              np.matmul(center_T_cam, translate_transform([ dx,
dy,  dz])),
              np.matmul(center_T_cam, translate_transform([ dx, -
dy,  dz])),
              np.matmul(center_T_cam, translate_transform([ dx,
dy, -dz])),
              np.matmul(center_T_cam, translate_transform([ dx, -
dy, -dz])),
              np.matmul(center_T_cam, translate_transform([-dx,
dy,  dz])),
              np.matmul(center_T_cam, translate_transform([-dx, -
dy,  dz])),
              np.matmul(center_T_cam, translate_transform([-dx,
dy, -dz])),
              np.matmul(center_T_cam, translate_transform([-dx, -
dy, -dz]))
        ]

  # constant transforms
  trackedPt_T_baselink = translate_transform([-0.058325, 0,
0.08125])
  colorCam_T_baselink = translate_transform([0.02, 0.033, 0.068])
  center_T_baselink = translate_transform([-0.015, 0.0, 0.003])
  baselink_T_trackedPt = inverse_transform(trackedPt_T_baselink)

  bag = rosbag.Bag(PATH_TO_INPUT_BAG)
  # subscribe to all /vrpn_client_node/carXX/pose for cars in
dataset
  # subscribe to '/carXX/camera/color/camera_info' and
'/carXX/camera/color/image_throttled' for camera car
  topics =
['/vrpn_client_node/car35/pose','/vrpn_client_node/car37/pose','
/vrpn_client_node/car38/pose','/car37/camera/color/camera_info',
'/car37/camera/color/image_throttled']

  ps_35 = []
  ps_37 = []
  ps_38 = []
  camInfo = None
  cam_37 = []

  for topic, msg, t in bag.read_messages(topics=topics):
        if topic == '/vrpn_client_node/car35/pose':
              ps_35.append((t, msg.pose))
        elif topic == '/vrpn_client_node/car37/pose':
```

```
            ps_37.append((t, msg.pose))
        elif topic == '/vrpn_client_node/car38/pose':
            ps_38.append((t, msg.pose))
        elif topic == '/car37/camera/color/camera_info':
            camInfo = msg
        elif topic == '/car37/camera/color/image_throttled':
            cam_37.append((t, msg))
```

Next, it will step over each video frame. Since the motion capture data is published at a much higher rate, we must find the mocap data to each video frame. Each message of a rostopic has a timestamp, so we pick the mocap message with the closest timestamp to the video frame's timestamp

```
 88        idx35 = 0
 89        idx37 = 0
 90        idx38 = 0
 91
 92        for idxImg in range(len(cam_37)):
 93            targetT = cam_37[idxImg][0]
 94            while idx35< len(ps_35)-1   and ps_35[idx35][0]   <
 95 targetT: idx35 += 1
 96            while idx37< len(ps_37)-1   and ps_37[idx37][0]   <
 97 targetT: idx37 += 1
 98            while idx38< len(ps_38)-1   and ps_38[idx38][0]   <
 99 targetT: idx38 += 1
100
101            # pick closest mocap data, not next
102            if idx35 > 0 and targetT - ps_35[idx35-1][0] <
103 ps_35[idx35][0] - targetT:
104                idx35 -= 1
105            if idx37 > 0 and targetT - ps_37[idx37-1][0] <
106 ps_37[idx37][0] - targetT:
107                idx37 -= 1
108            if idx38 > 0 and targetT - ps_38[idx38-1][0] <
109 ps_38[idx38][0] - targetT:
110                idx38 -= 1
111
112            # unwrap mocap pose into position, orientation
113            pos35 = ps_35[idx35][1].position
               ori35 = ps_35[idx35][1].orientation
               pos37 = ps_37[idx37][1].position
               ori37 = ps_37[idx37][1].orientation
               pos38 = ps_38[idx38][1].position
               ori38 = ps_38[idx38][1].orientation
```

Next, we create and use transforms to identify where the other cars' corners are on the frame. We load in the frame as an OpenCV image. Should you need to debug, you can draw the points of interest on the frame.

```
113     # create Transforms for TrackedPts w.r.t. World
114     pt35_T_w = transformerROS.fromTranslationRotation((pos35.x,
115 pos35.y, pos35.z), (ori35.x, ori35.y, ori35.z, ori35.w))
116     pt37_T_w = transformerROS.fromTranslationRotation((pos37.x,
117 pos37.y, pos37.z), (ori37.x, ori37.y, ori37.z, ori37.w))
118     pt38_T_w = transformerROS.fromTranslationRotation((pos38.x,
119 pos38.y, pos38.z), (ori38.x, ori38.y, ori38.z, ori38.w))
120
121     # create Transforms for points of interest w.r.t. World
122     base37_T_w = np.matmul(pt37_T_w, baselink_T_trackedPt)
123     cam37_T_w = np.matmul(base37_T_w, colorCam_T_baselink)
124
125     base35_T_w = np.matmul(pt35_T_w, baselink_T_trackedPt)
126     center35_T_w = np.matmul(base35_T_w, center_T_baselink)
127     base38_T_w = np.matmul(pt38_T_w, baselink_T_trackedPt)
128     center38_T_w = np.matmul(base38_T_w, center_T_baselink)
129
130     # create Transforms for points of interest w.r.t. Camera
131     w_T_cam37 = inverse_transform(cam37_T_w)
132     center38_T_cam37 = np.matmul(w_T_cam37, center38_T_w)
133     center35_T_cam37 = np.matmul(w_T_cam37, center35_T_w)
134
        # create Transforms for bounding box corners
        corners35 = get_corners(center35_T_cam37)
        corners38 = get_corners(center38_T_cam37)
```

Once we've found where each bounding box is, we have to decide whether or not to include it in the dataset. First we reject all labels of cars behind the camera. We also wish to throw out a label when one car is occluding another - we only want the front car to be labeled in this case. Finally we save the image and its label in their corresponding locations.

```
135     # convert to OpenCV image
136     car37_image = bridge.imgmsg_to_cv2(cam_37[idxImg][1], "bgr8")
137
138     # find pixel coordinates of centroids
139     # the Camera Pinhole model uses +x right, +y down, +z forward
140     center35_3d = (-center35_T_cam37[1,3], -
141 center35_T_cam37[2,3], center35_T_cam37[0,3])
142     center38_3d = (-center38_T_cam37[1,3], -
143 center38_T_cam37[2,3], center38_T_cam37[0,3])
144     camModel.fromCameraInfo(camInfo)
145     center35_2d = camModel.project3dToPixel(center35_3d)
```

```
146     center38_2d = camModel.project3dToPixel(center38_3d)
147     center35_round = (int(center35_2d[0]), int(center35_2d[1]))
148     center38_round = (int(center38_2d[0]), int(center38_2d[1]))
149 
150 
151     # only draw on frame if the observed robot is behind the
152 camera
153     car35_rect = None
154     if center35_3d[2] > 0:
155         # draw centroid
156         cv2.circle(car37_image, center35_round, 3, (0,255,0), 2)
157 
158         # initialize rectangle bounds to image size
159         # cv's image.shape is formatted as (height, width,
160 length) a.k.a. (y, x, z)
161         xmin = car37_image.shape[1] + 1
162         xmax = -1
163         ymin = car37_image.shape[0] + 1
164         ymax = -1
165 
166         for corneridx, corner in enumerate(corners35):
167             # find pixel coordinates of corners
168             corner_3d = (-corner[1,3], -corner[2,3], corner[0,3])
169             corner_2d = camModel.project3dToPixel(corner_3d)
170             corner_round = (int(corner_2d[0]), int(corner_2d[1]))
171             # draw the front (first 4) corners in different
172 colors
173             color = (255,255,0)
174             cv2.circle(car37_image, corner_round, 2, color, 1)
175 
176             xmin = min(xmin, corner_round[0])
177             xmax = max(xmax, corner_round[0])
178             ymin = min(ymin, corner_round[1])
179             ymax = max(ymax, corner_round[1])
180 
181         # save rectangle
182         car35_rect = ((xmin, ymin), (xmax, ymax))
183     car38_rect = None
184     if center38_3d[2] > 0:
185         # draw centroid
186         cv2.circle(car37_image, center38_round, 3, (0,255,0), 2)
187 
188         # cv's image.shape is formatted as (height, width,
189 length) a.k.a. (y, x, z)
190         xmin = car37_image.shape[1] + 1
191         xmax = -1
192         ymin = car37_image.shape[0] + 1
193         ymax = -1
```

```
194
195         for corneridx, corner in enumerate(corners38):
196             # find pixel coordinates of corners
197             corner_3d = (-corner[1,3], -corner[2,3], corner[0,3])
198             corner_2d = camModel.project3dToPixel(corner_3d)
199             corner_round = (int(corner_2d[0]), int(corner_2d[1]))
200             # draw the front (first 4) corners in different
201 colors
202             color = (0,255,255)
203             cv2.circle(car37_image, corner_round, 2, color, 1)
204
205             xmin = min(xmin, corner_round[0])
206             xmax = max(xmax, corner_round[0])
207             ymin = min(ymin, corner_round[1])
208             ymax = max(ymax, corner_round[1])
209
210         # save rectangle
211         car38_rect = ((xmin, ymin), (xmax, ymax))
212
213     # draw rectangles & check for overlap
214     overlap_tolerance = 10
215     if car35_rect is not None:
216         if car38_rect is not None:
217             if car35_rect[0][0] > car38_rect[0][0] -
218 overlap_tolerance and car35_rect[0][1] > car38_rect[0][1] -
219 overlap_tolerance and \
220                 car35_rect[1][0] < car38_rect[1][0] +
221 overlap_tolerance and car35_rect[1][1] < car38_rect[1][1] +
222 overlap_tolerance:
223                     # reject draw
224                     print('car38 overlaps car35')
225             else:
226                 cv2.rectangle(car37_image, car35_rect[0],
227 car35_rect[1], (0,255,0), 1)
228         else:
229             cv2.rectangle(car37_image, car35_rect[0],
230 car35_rect[1], (0,255,0), 1)
231
232     if car38_rect is not None:
233         if car35_rect is not None:
234             if car38_rect[0][0] > car35_rect[0][0] -
235 overlap_tolerance and car38_rect[0][1] > car35_rect[0][1] -
    overlap_tolerance and \
                 car38_rect[1][0] < car35_rect[1][0] +
    overlap_tolerance and car38_rect[1][1] < car35_rect[1][1] +
    overlap_tolerance:
                     # reject draw
                     print('car35 overlaps car38')
```

```
                else:
                    cv2.rectangle(car37_image, car38_rect[0],
    car38_rect[1], (0,0,255), 1)
            else:
                cv2.rectangle(car37_image, car38_rect[0],
    car38_rect[1], (0,0,255), 1)


        # Add frame to video
        vid_out.write(car37_image.astype('uint8'))

          # end video
          vid_out.release()
```

v. Setting Up YOLOv5

First clone the YOLOv5 repo: – https://github.com/ultralytics/yolov5
Now install all the requirements from requirements.txt Now create your train/valid/test files. Each file will contain the location of the images you will use for training (the labels will be automatically grabbed since the directory structure is known ahead of time). We split our dataset in a 70/20/10 ratio for train/valid/test, respectively.

Here are the first 3 lines of our train.txt:

- /mnt/hard_data/carpose/dataset/images/car37_longtrim_000000.jpg
- /mnt/hard_data/carpose/dataset/images/car37_longtrim_000001.jpg
- /mnt/hard_data/carpose/dataset/images/car37_longtrim_000002.jpg

Next create carpose.yaml, which will describe the dataset. Fill in the paths to your train/valid/test files. We only have 1 class: Car.

Here is our carpose.yaml file:

```
# train and val datasets (image directory or *.txt file with image paths)
train: /home/ugrads/WRK/carpose/src/yolov5/data/train.txt
val: /home/ugrads/WRK/carpose/src/yolov5/data/valid.txt
test: /home/ugrads/WRK/carpose/src/yolov5/data/test.txt
# number of classes in your dataset
nc: 1
# class names
names: ['Car']
```

This file should be in yolov5/data.

vi. Training Your Model

Now we can begin training our model! YOLOv5 has four different model sizes: S, M, L, and XL. We used the smallest: yolov5s.yaml. The parameters we used were: python train.py –data ./data/carpose.yaml –cfg ./models/yolov5s.yaml –logdir /mnt/hard_data/Checkpoints/carpose/runs/ –workers 0 –img-size 640 –single-cls –device 1 –batch 16 –epochs 10 –weights ''

Change the logdir to wherever you want the output to be, this will include model checkpoints, plots, and examples from each epoch. The –workers parameter only worked when set to 0 for us. The –device parameter selects which cuda device to use. You can increase/decrease the batch size based on your GPU, and change the training length (epochs). You can also use previously stored weights by filling in the –weights parameter.

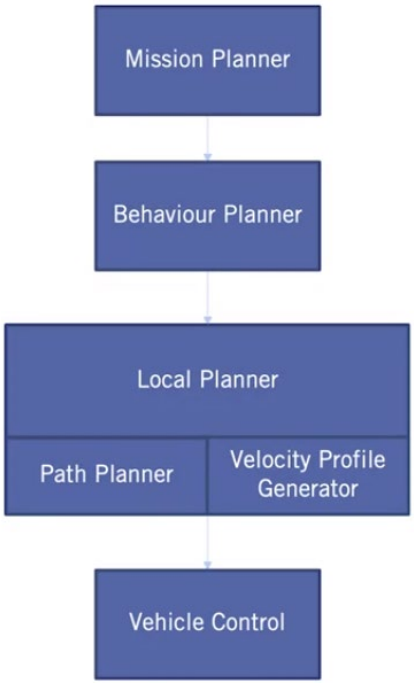
Alternatively, you can download our trained model.

vii. Demo Video

IV. Motion Planning

A. Introduction

For any type of autonomous robot, motion planning adds the essential piece in the autonomy. It helps to find a sequence of valid configurations that moves the vehicle from a source location to a destination. Motion planning algorithms help to plan the shortest obstacle-free path to the goal.



We can classify motion planning into these categories:

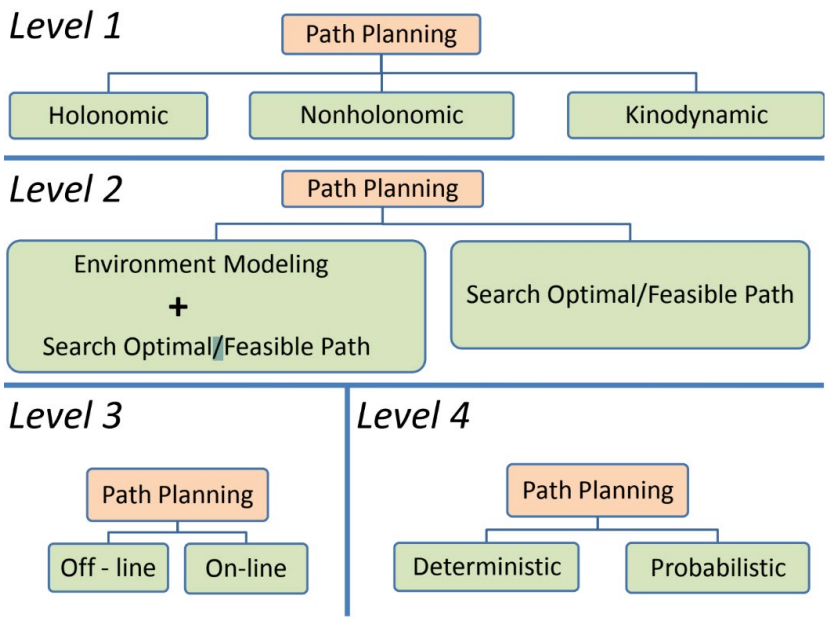
- Global Mission Planning: Path planning algorithms

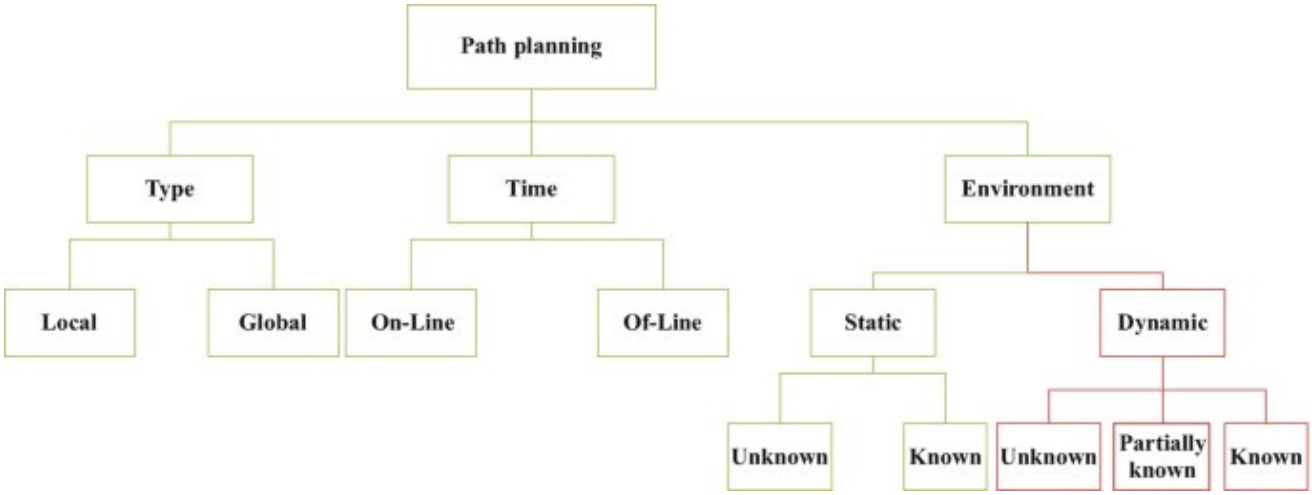
- Behavioral Planning: High-level decision making

- Local Re-Planning: Trajectory generation and optimization

- Velocity Planning (Longitudinal Control): Velocity profile generation

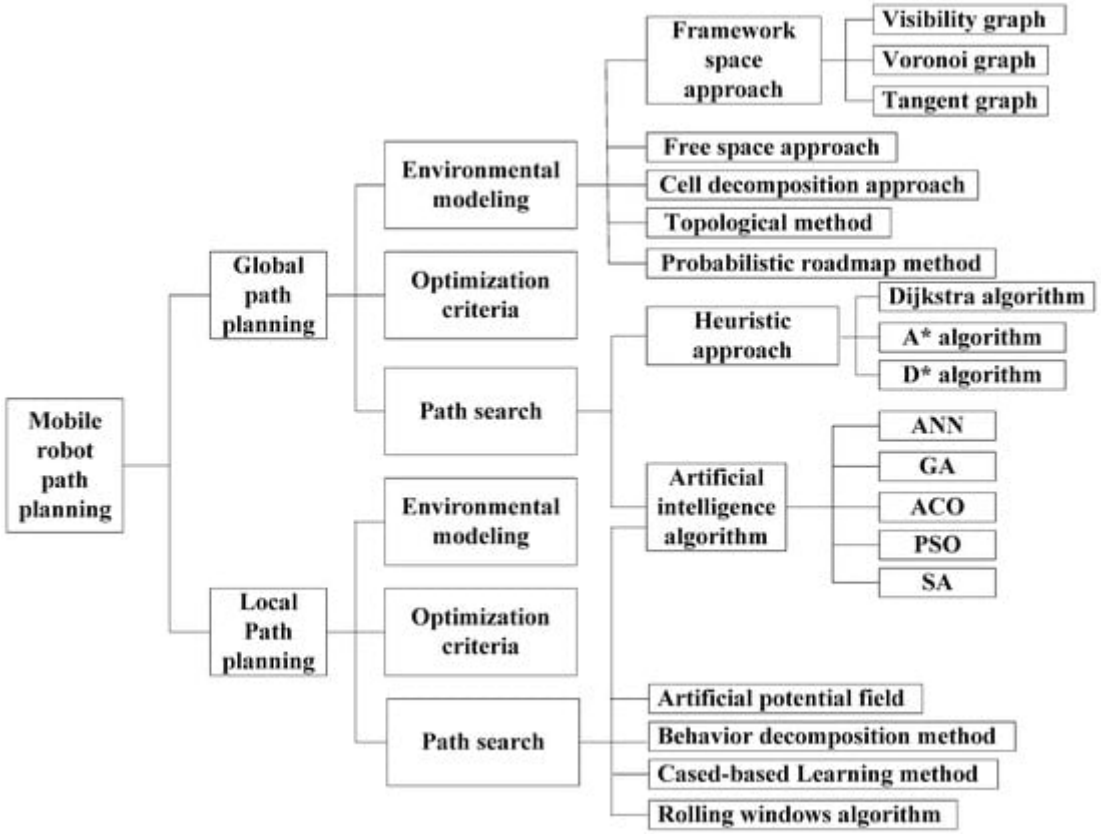
Classification of Path Planning Algorithms





Criterion	Category	Overview	Suitability
Changes before or after task execution	Offline path planning	Before the task is executed, the optimal path is generated and does not change until the task is completed	Simple; Stable; Universal
	Online path planning	After the task is executed, the path can be re-planned according to the situation	Complicated; Unstable
Environmental change	Static environmental path planning	Environmental information no longer changes	Simple; Stable; Universal
	Dynamic environmental path planning	Environmental information changes in real time	Complex; Not suitable for structured links
Environmental perception	Global path planning	Planning for global tasks	Overall task
	Local path planning	Planning two-point paths with known local environmental information	Local two points

Local path planning	Global path planning
Sensor-based	Map-based
Reactive navigation	Deliberative navigation
Fast response	Relatively slower response
Suppose that the workspace area is incomplete or partially incomplete	The workspace area is known
Generate the path and moving toward target while avoiding obstacles or objects	Generate a feasible path before moving toward the goal position
Done online	Done offline



Resources:

A Consolidated Review of Path Planning and Optimization Techniques: Technical Perspectives and Future Directions: https://www.mdpi.com/2079-9292/10/18/2250/htm

Path Planning for Autonomous Mobile Robots: A Review: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8659900/

Book: http://lavalle.pl/planning/booka4.pdf

B. Global Mission Planner: Path planning algorithms

1. Overview

In a typical autonomous vehicle workflow, the perception model provides information on how the environment around the vehicle looks like, that includes the obstacles that the robot should avoid bumping into. We get the location of the robot at each movement or state from the localization algorithms.

For global path planning in the case of non-holonomic vehicles, we can classify the types of path planning algorithms into:

- Heuristic Sampling-based algorithms such as: RRT, RRT* etc

- Search-based algorithms such as Hybrid A* etc

Search-based planning is a motion planning method which uses graph search methods to compute paths or trajectories over a discrete representation of the problem. Because a

graph is inherently discrete, all graph-search algorithms require this discrete representation. Search-based planning can then be seen as two problems: how to turn the problem into a graph, and how to search the graph to find the best solution.

This is in contrast to sample-based planning methods, which are able to solve problems in continuous space without a graph representation (though certain sampling-based methods do discretize the space). While the pros and cons between the two methods can be debated for many hours, the most apparent difference between the two methods are speed and optimality: graph-search methods can guarantee how "efficient" a solution is (defined more later), but, in general, does so at the expense of computation time. Sampling based planners run fast, but can result in unusual looking and possibly inefficient paths.

2. Simple Trajectory Planning

Create a simple ROS node to read commands (velocity and steering angle) from a file line by line, and send them to the simulator to be applied to the simulated car. Each line will denote a command to be applied for one second. The first line is a message to send as the “starting pose” of the car.

The input files will be of the form:

```
0,0,0.0
2.0,0.09
3.0,-0.15
```

The first line is the initial position, of the form x, y, theta, where x and y are the starting coordinates in the map, and theta is the initial angle of the car. The following two lines describe commands that tell the car how fast to go, and at what steering angle. The first says to run at 2.0 meters per second, with a steering angle of 0.09 radians. The second, to run at 3.0 meters per second, with a steering angle of -0.15 radians. We will be applying each command for 1 second. Note that a positive steering angle corresponds to a left turn and a negative steering angle corresponds to a right turn.

Examples:

plans/straight_line.txt

```
0,0,0
2,0.0
3,0.0
4,0.0
5,0.0
6,0.0
6,0.0
6,0.0
5,0.0
4,0.0
3,0.0
2,0.0
```

plans/figure_8.txt

```
0,0,0.785
2.0,0.09
2.0,0.09
2.0,0.09
2.0,0.09
2.0,0.09
2.0,0.09
2.0,0.09
2.0,0.09
2.0,0.09
2.0,0.09
2.0,0.09
2.0,-0.09
2.0,-0.09
2.0,-0.09
2.0,-0.09
2.0,-0.09
2.0,-0.09
2.0,-0.09
2.0,-0.09
2.0,-0.09
2.0,-0.09
2.0,-0.09
```

Code: `src/path_publisher.py`

Explanation: https://mushr.io/tutorials/intro-to-ros/

```python
#!/usr/bin/env python

import rospy
from ackermann_msgs.msg import AckermannDrive, AckermannDriveStamped
from geometry_msgs.msg import (
    Point,
    Pose,
    PoseWithCovariance,
    PoseWithCovarianceStamped,
    Quaternion,
)
from tf.transformations import quaternion_from_euler


def run_plan(pub_init_pose, pub_controls, plan):
    init = plan.pop(0)
    send_init_pose(pub_init_pose, init)

    for c in plan:
        send_command(pub_controls, c)

```

```
22
23 def send_init_pose(pub_init_pose, init_pose):
24     pose_data = init_pose.split(",")
25     assert len(pose_data) == 3
26
27     x, y, theta = float(pose_data[0]), float(pose_data[1]),
28 float(pose_data[2])
29     q = Quaternion(*quaternion_from_euler(0, 0, theta))
30     point = Point(x=x, y=y)
31     pose = PoseWithCovariance(pose=Pose(position=point,
32 orientation=q))
33     pub_init_pose.publish(PoseWithCovarianceStamped(pose=pose))
34
35
36 def send_command(pub_controls, c):
37     cmd = c.split(",")
38     assert len(cmd) == 2
39     v, delta = float(cmd[0]), float(cmd[1])
40
41     dur = rospy.Duration(1.0)
42     rate = rospy.Rate(10)
43     start = rospy.Time.now()
44
45     drive = AckermannDrive(steering_angle=delta, speed=v)
46
47     while rospy.Time.now() - start < dur:
48         pub_controls.publish(AckermannDriveStamped(drive=drive))
49         rate.sleep()
50
51
52 if __name__ == "__main__":
53     rospy.init_node("path_publisher")
54
55     control_topic = rospy.get_param("~control_topic",
56 "/car/mux/ackermann_cmd_mux/input/navigation")
57     pub_controls = rospy.Publisher(control_topic,
58 AckermannDriveStamped, queue_size=1)
59
60     init_pose_topic = rospy.get_param("~init_pose_topic",
61 "/initialpose")
62     pub_init_pose = rospy.Publisher(init_pose_topic,
63 PoseWithCovarianceStamped, queue_size=1)
64
65     plan_file = rospy.get_param("~plan_file")
66
67     with open(plan_file) as f:
         plan = f.readlines()
```

```
    # Publishers sometimes need a warm-up time, you can also wait
until there
    # are subscribers to start publishing see publisher
documentation.
    rospy.sleep(1.0)
    run_plan(pub_init_pose, pub_controls, plan)
```

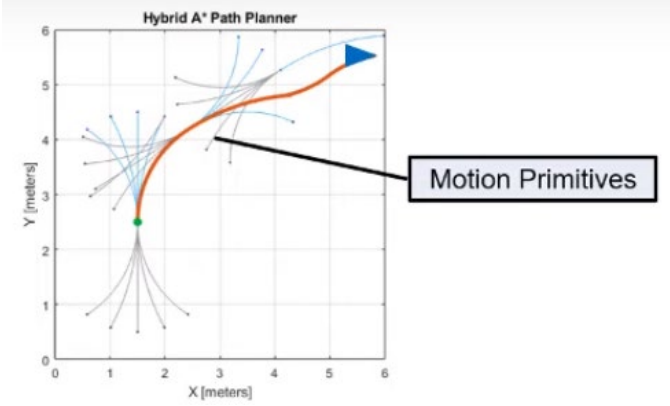
3. Graph Search-based Methods

i. Search-Based Planning Library (SBPL) http://www.sbpl.net/

ii. For Non-holonomic Robots

- **Hybrid A***

Hybrid A* is similar to the traditional A* algorithm (which can only be used for holonomic robots) in how the search space, that is (x,y,θ), is discretized. But unlike A*, hybrid A* associates a continues 3-d state of the vehicle, or as we call them motion primitives with each grid cell. These motion primitives are checked for collisions in the map using state validators which generates a smooth obstacle-free path.

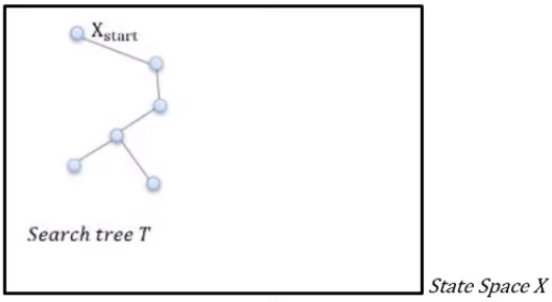


Hybrid A* also guarantees kinematic feasibility and takes differential constraints (orientation and velocity) into account. We can give the orientation of the vehicle as an input along with the start and goal locations and we can see how we get different drivable paths based on different goal orientations.

A big problem with grids and search-based algorithms is the exponential explosion in the number of nodes when the dimensionality of the problem increases and the time to find a solution quickly becomes intractable. Search-based algorithms are not suitable for applications that have high degrees of freedoms or when the map size is very large. Storing the grid information for a large map becomes computationally expensive and this is where the sampling-based planning algorithms are helpful.

- **Rapidly-exploring Random Tree (RRT)**

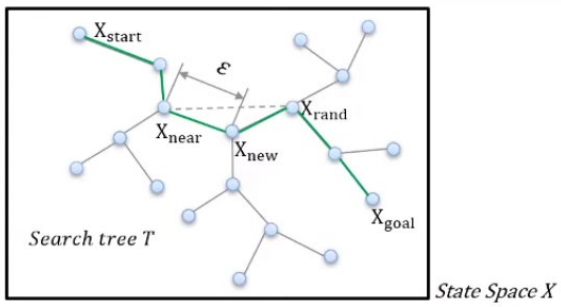
RRT creates a tree in the state space with randomly sampled states or nodes. The RRT algorithm is designed for efficiently searching non-convex high dimensional spaces. RRTs are constructed incrementally in a way that it quickly reduces the expected distance of a randomly chosen node in the tree.



We first define the state space with an initial tree T that has start point as one of the nodes in it to represent the map in a way that the planning algorithm or the planner can understand it we represent it in the form of a state space.

A state of our vehicle is its position (x,y), its orientation (theta), and its steering angle. A state space is a set of many possible vehicle states.

For example, we randomly sample a state Xrand in the state space X, and we find a node Xnear nearest to Xrand that already exists in the tree. This Xrand can be anywhere in the state space, so we need another node Xnew between Xrand and Xnear to expand the tree. And we repeat this process until we reach the Xgoal. Every time we sample a new load node like Xnew we also check for collision between the nodes Xrand and Xnew.



We can use motion primitives or motion models like Dubins-curve to generate local motion from one node to another RRTs are particularly suitable for path planning problems that involve obstacles and differential constraints.

The RRT algorithm gives a valid path but not necessarily the shortest path which brings us to the following algorithms.

- **ARA* planning algorithm**
- **MuSHR Global Planner Node**

This node generates a plan that the RHC controller will follow. The planner does not consider dynamic obstacles when constructing its plan; rather it uses a static map of the environment. It chains the car's motion primitives together into a plan, using a search algorithm (such as A*).

mushr_gp: Global Planner

This ROS package hosts the global planner for the MuSHR system. It wraps a search-based planner from SBPL (Search-Based Planning Library). Search based methods have two parts - first, the

planning problem and environment must be represented by a graph, and then the graph must be searched from a starting point to a goal.
This MuSHR global planner builds upon the SBPL repo by Search-Based Planning Lab at Carnegie Mellon University. MuSHR specifically makes use of the ARA* planning algorithm and plans on an implicit graph in a SE2 (x, y, θ) state space.

Publishers

Topic	Type	Description
/path_topic	geometry_msgs/Path	The trajectory computed by the planner.

Subscribers

Topic	Type	Description
/map	nav_msgs/OccupancyGrid	Uses the provided occupancy grid as the graph for planning.
/goal_topic	geometry_msgs/PoseStamped	Goal to compute path to.
/start_topic	geometry_msgs/PoseStamped	Starting location of the path being computed.

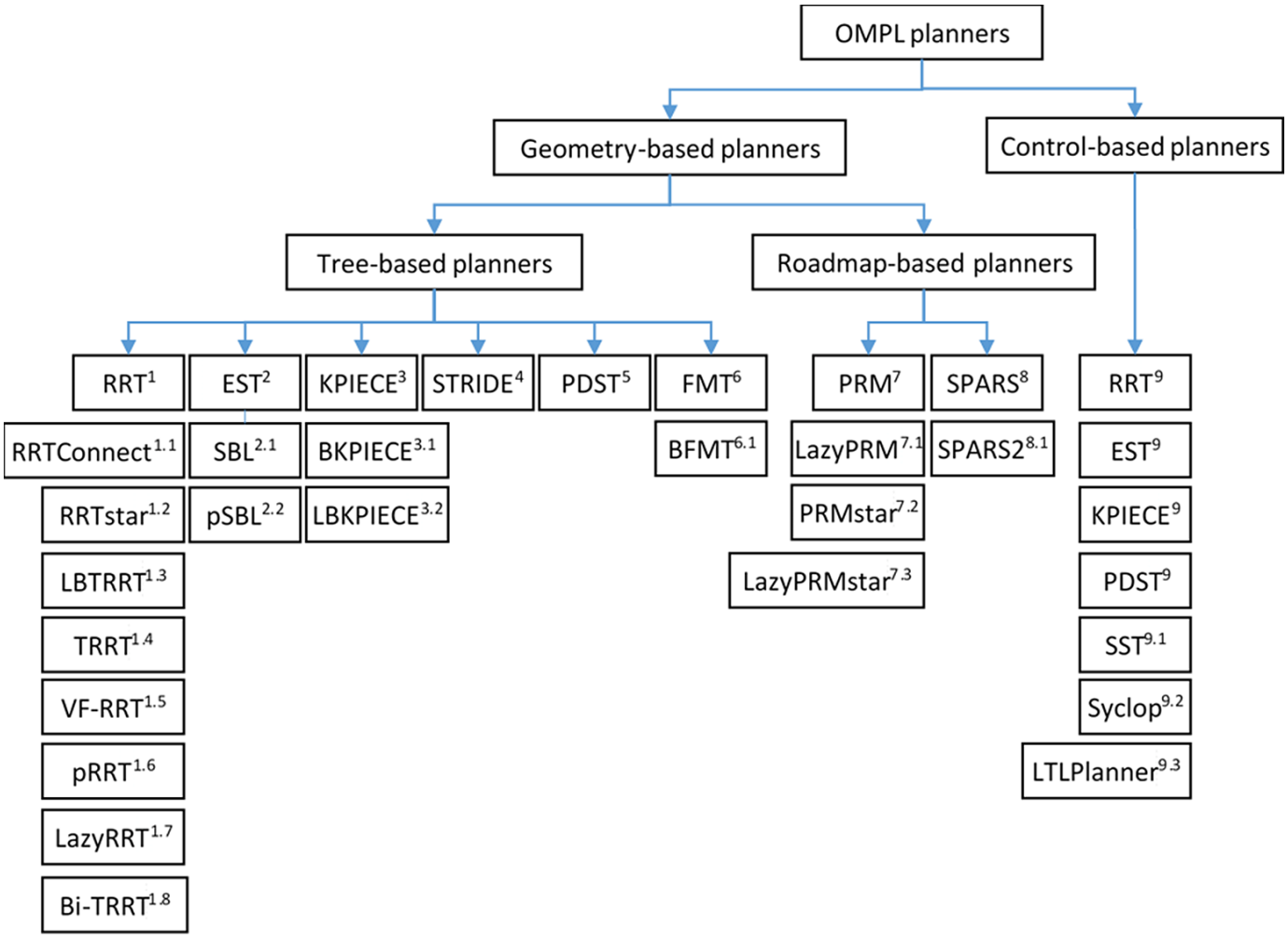
NOTE: The reason why there are two planners (mushr_gp and mushr_gprm) is because mushr_gp is too resource intensive to be run on the jetson nano 4GB variant. However, if the desktop/laptop computer remains connected and in range of the MuSHR car, you can run mushr_gp on the computer instead as they share the same ROS master. If you need the planner to run on the jetson nano, we recommend using the mushr_gprm package. For the Jetson Xavier NX or when running on the sim exclusively, mushr_gp will work. Both repositories contain ROS packages that reproduce the desired functionality. You need only concern yourself with each package's launch files to use them effectively. You can find the launch files in each package's launch directory.

iii. For Holonomic Robots

4. Heuristic Sampling-based Methods

i. Open Motion Planning Library (OMPL)

https://ompl.kavrakilab.org/



Source: https://www.cambridge.org/core/journals/robotica/article/abs/comparison-of-different-samplebased-motion-planning-methods-in-redundant-robotic-manipulators/823E7554AFF9B8D43890BCC67BD9083C

ii. For Non-Holonomic Robots

- **RRT-A***

RRT-A* is an improved heuristic of RRT where the cost function of A* is introduced into the RRT algorithm to optimize the performance. Meanwhile, several metric functions are used as the heuristic information functions respectively to measure the performance of different metric function. Research and simulation results have shown* that the Manhattan heuristic information function based RRT-A* planning algorithm is better than the other improved RRT algorithms in optimization path and computational cost.

When choosing the nearest neighbor Xnear as the following best input variable, the original RRT adopts the principle of nearest neighbor accomplished by the metric function. And there are several metric functions used in RRT algorithm as follows, only in the space of two dimensions.

The Manhattan distance between two points can be expressed as:

$$d_{\text{Manhattan}}(i, j) = |x_{i1} - x_{j1}| + |x_{i2} - x_{j2}|$$

- **BIT***
- **ABIT***

iii. **For Holonomic Robots**

5. **Reactive Methods (reactive = with obstacle avoidance):**

https://docs.google.com/presentation/d/1SL5zVclQy-PI55SdyWugdKtCU1E7OzxIu8Uuz6nfaWg/edit

i. **Follow the Gap**

Naïve Follow the Gap

UNC Follow the Gap

F1/10 Follow the Gap

ii. **Bug Algorithms**

iii. **TangentBug Algorithms**

iv. **Artificial Potential Fields**

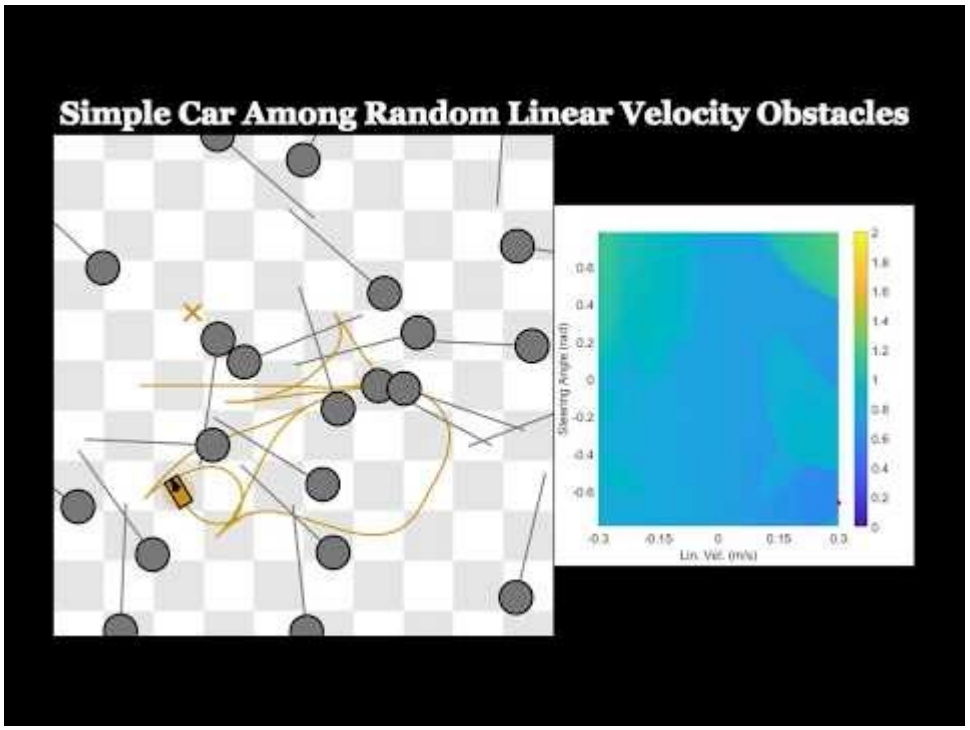
v. **Gradient Descent**

vi. **Multi-Agent Reactive Navigation**

NH-TTC: A generalized framework for anticipatory collision avoidance

A navigation system enables a robot to move quickly between different poses while avoiding collisions with the environment or other agents. The navigation system in this MuSHR project uses a slightly modified version of the NH-TTC system. It is a decentralized, multi-agent navigation system that considers the Non-holonomic constraints of the car and the time to collision when finding the optimal control actions. To learn more about how it works, check out the paper here!

NH-TTC is a general method for fast, anticipatory collision avoidance for autonomous robots having arbitrary equations of motions. Our proposed approach exploits implicit differentiation and subgradient descent to locally optimize the non-convex and non-smooth cost functions that arise from planning over the anticipated future positions of nearby obstacles. The result is a flexible framework capable of supporting high-quality, collision-free navigation with a wide variety of robot motion models in various challenging scenarios. We show results for different navigating tasks, with our method controlling various numbers of agents (with and without reciprocity), on both physical differential drive robots, and simulated robots with different motion models and kinematic and dynamic constraints, including acceleration-controlled agents, differential-drive agents, and smooth car-like agents. The resulting paths are high quality and collision-free, while needing only a few milliseconds of computation as part of an integrated sense-plan-act navigation loop.



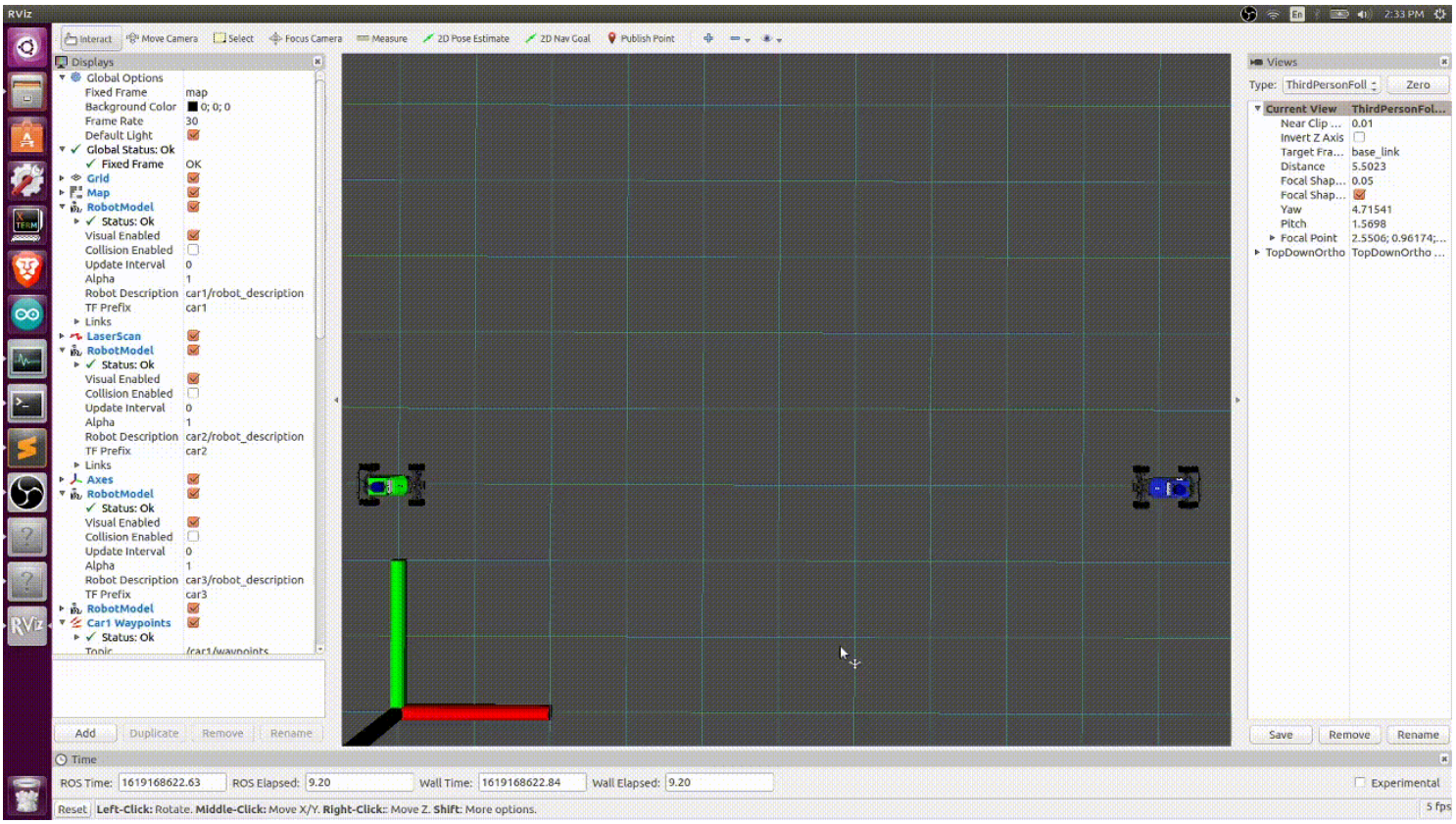
Code: https://github.com/davisbo/NHTTC

Paper: http://motion.cs.umn.edu/r/NH-TTC/arxiv-NHTTC.pdf

Video: https://youtu.be/9EG48v0r_YY

Running the Simulator example:

In this demo, the blue car is trying to follow the blue arrows and the green car is trying to follow the green arrows. The green car is supposed to stop a small distance after crossing the intersection of the two paths whereas the blue car is supposed to move towards the top after the intersection of the two paths. The two paths coincide to force the navigation system to display its capabilities



The system takes a set of waypoints rather than a single goal point. The message to publish for this is:/car_name/waypoints of type: geometry_msgs/PoseArray.

The waypoint can also contain just one waypoint, so it is possible to test the system with single waypoints if you prefer that. The reason why taking an array is preferred is so that the waypoint management code doesn't have to be written by you (user). You simply pass the set of the waypoints in and the navigation sytem takes care of managing them on it's own.

The z axis coordinate represents the time difference between 2 waypoints.

Note that a z axis value of 0.001 equals a time difference of 1 unit. This is done so that the waypoints don't look like they're floating off the ground when visualized in rviz. The unit of time is equal to the time it takes for the car to cover the distance between two waypoints in a straight line at the rated speed. The reason for this is to allow the system's speed to be scaled up or down without changing the global plan's timing itself.

Tuning the parameters

The multi-agent navigation system *can* work out of the box for most applications, however, it is possible to tune it. The parameters can be changed inside the yaml config file, in this case, the nhttc_demo.yaml. The navigation system uses a solver which has some level of stochasticity to it, which can lead to slightly different behavior. The demonstrations shown here are to explain the

effect of changing the parameters and do not indicate the exact performance. Also note that there may be additional parameters besides the ones specified here. Those are experimental and should be set to false or left as is by the user.

carrot_goal_ratio: 1.0

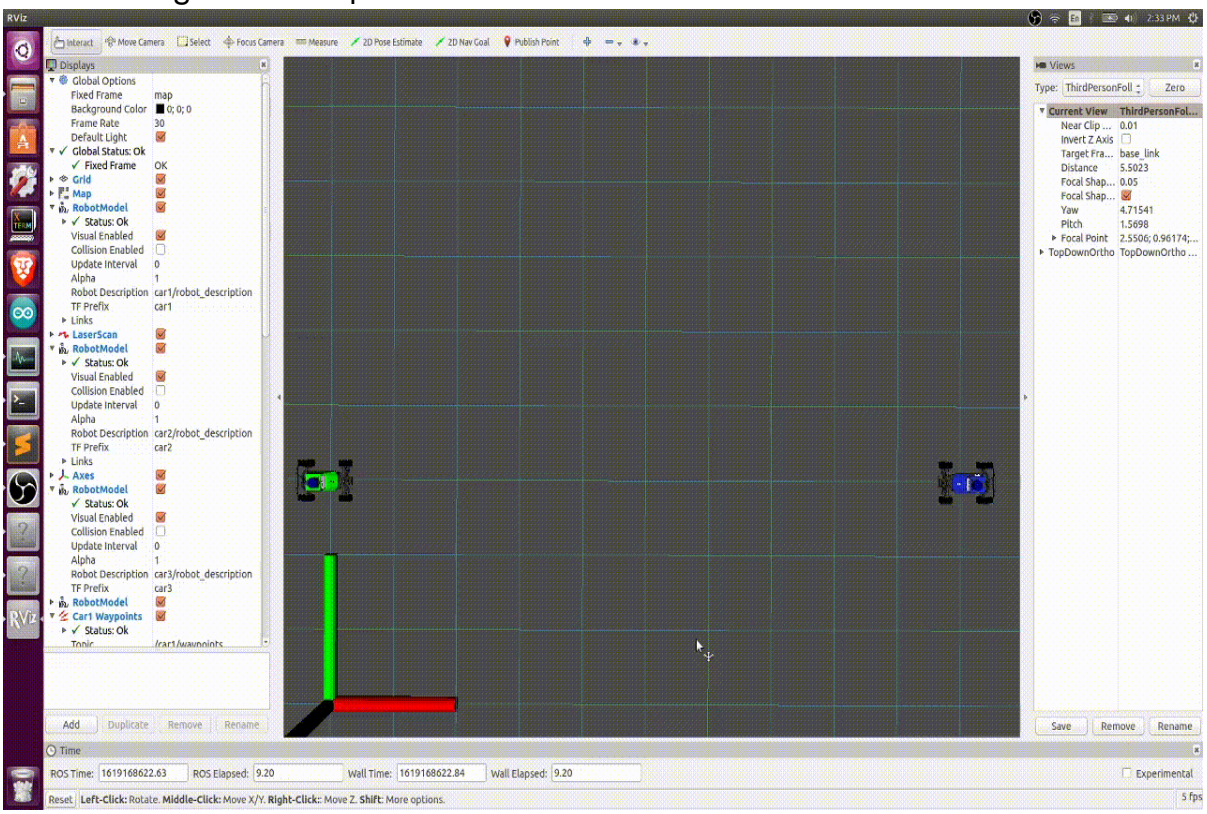
max_ttc: 6.0

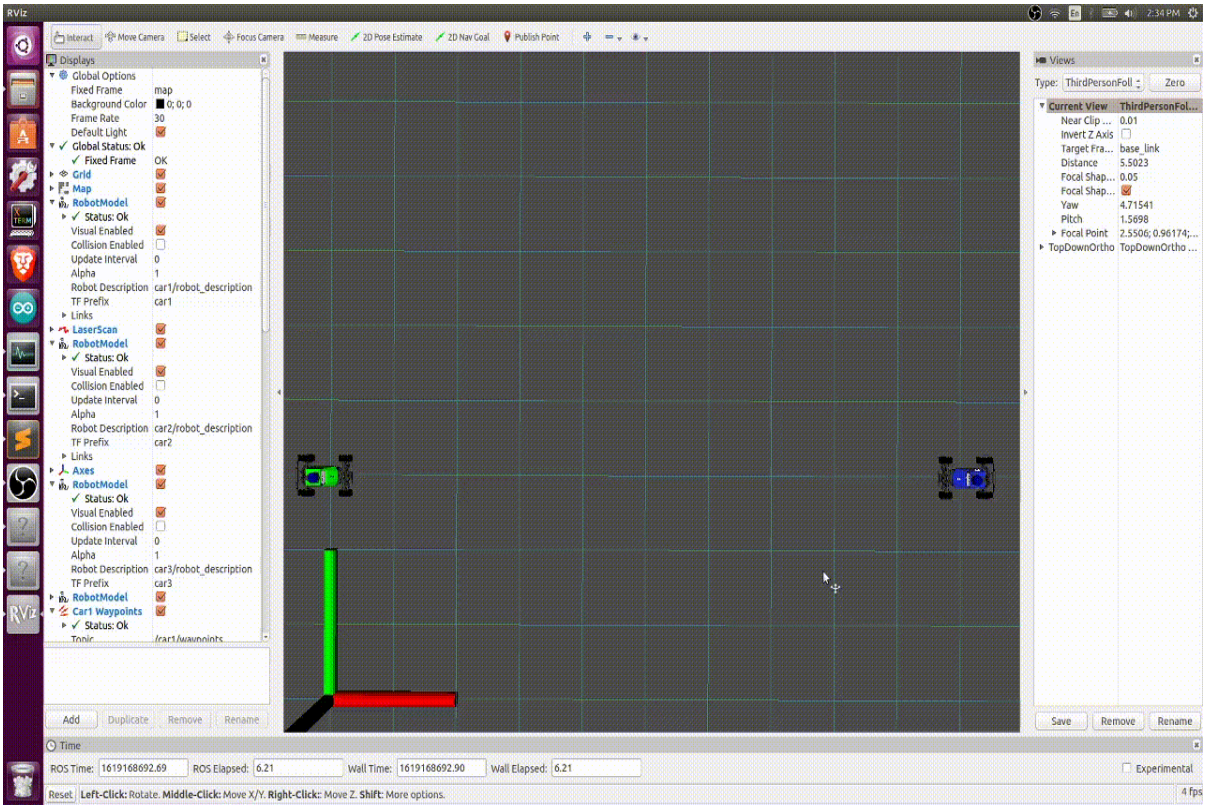
solver_time: 20

obey_time: true

1) carrot_goal_ratio: The ROS wrapper implements a carrot-goal navigation system where waypoints are selected from a prescribed path. The waypoints are selected such that they are some “lookahead” distance away from the car. Keeping the car aimed at a waypoint farther away prevents it from getting stuck in a local minimum. Keeping this lookahead closer to the car makes sure the car does not deviate too far away from the prescribed path while getting to the point farther down the line. The ratio of this lookahead distance or carrot-goal distance to the turning radius has been defined as the carrot-goal ratio. The reason for that name is that the way the system works is akin to a donkey(MuSHR car) trying to eat a carrot(waypoint) hung from a stick that you(navigation system) are holding while sitting on top of it. The donkey moves where the carrot goes but can never actually reach it:

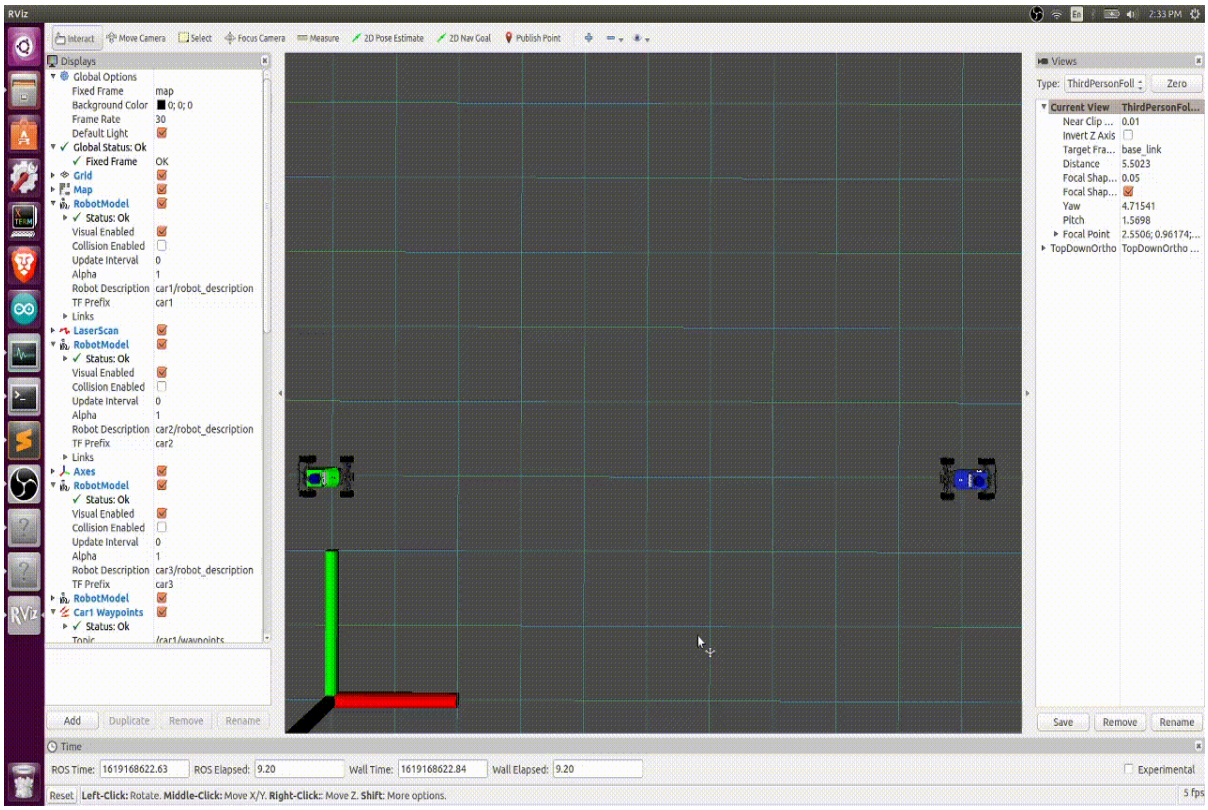
A value of 1.0 means the carrot goal distance is the same as the turning radius. A value of 2 indicates that the carrot goal distance is twice the turning radius. larger numbers result in smoother navigation, however, they come with the drawback of greater path-deviation as the system will tend to “round off” corners a lot sooner. The first figure shows the performance with carrot-goal-ratio of 1.0, and the second figure shows performance with a ratio of 1.5:

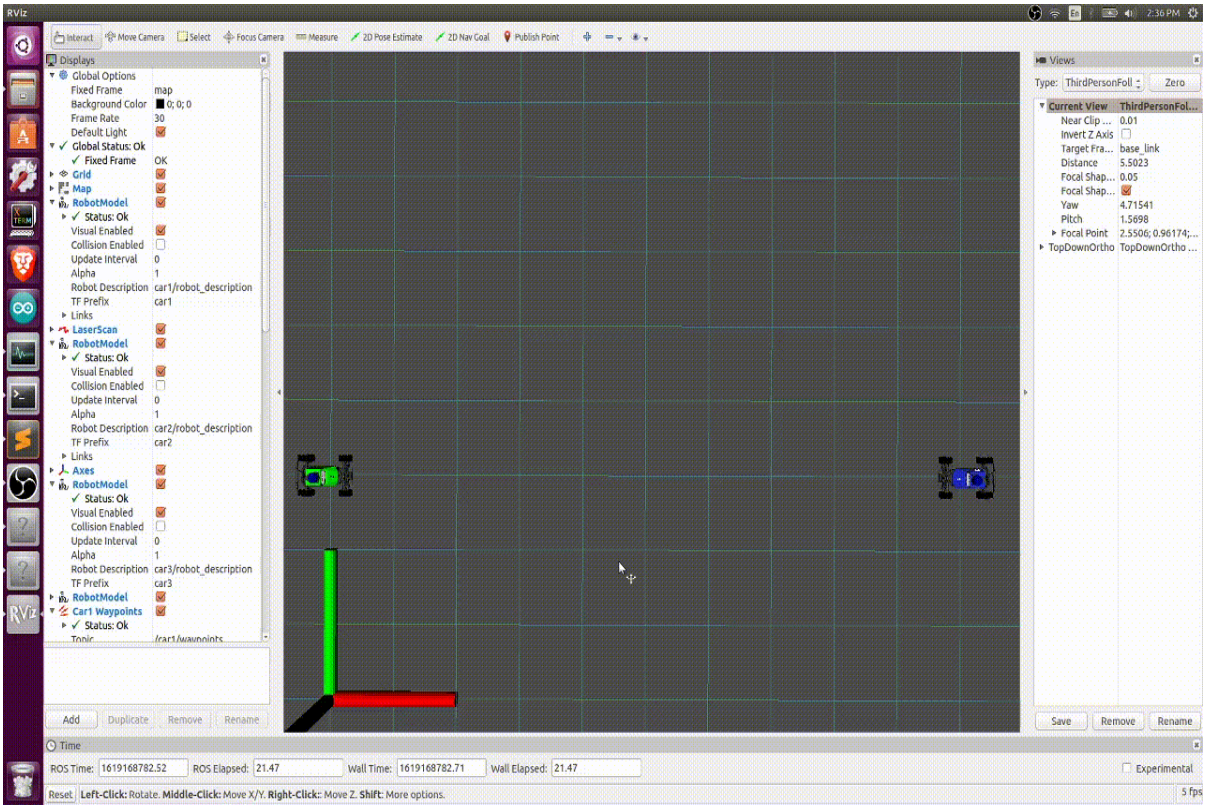




If the car tends to get stuck around turns, increase the carrot-goal ratio in increments of 0.1. If the car appears to be rounding off the turns too soon or significantly deviating from the path near turns, causing issues with other agents, reduce the carrot-goal-ratio in decrements of 0.1.

2) max_ttc: Stands for maximum time to collision. This parameter decides which agents to consider and which to not consider when optimizing for the next control action. The time to collision is calculated using the current state (pose as well as twist) of all agents. A larger max_ttc results in a larger time horizon for considering potential collisions. A larger max_ttc will make the car respond earlier to other agents but can result in the car deviating from its path too early. A smaller max_ttc will make the car less sensitive to agents far away but may result in the car responding too late and ending up in deadlocks more often. The first figure shows the performance with a max_ttc of 3.0 and the second shows the performance with max_ttc of 6.0 seconds. Notice how the cars start avoiding each other earlier in the second example.

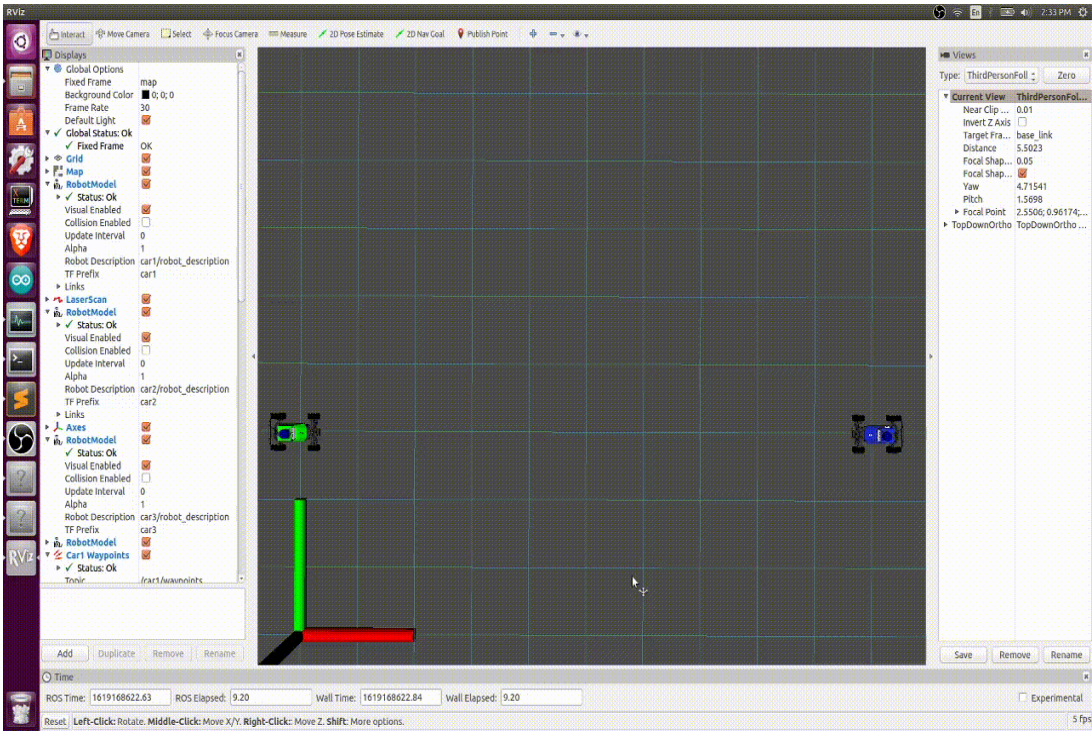


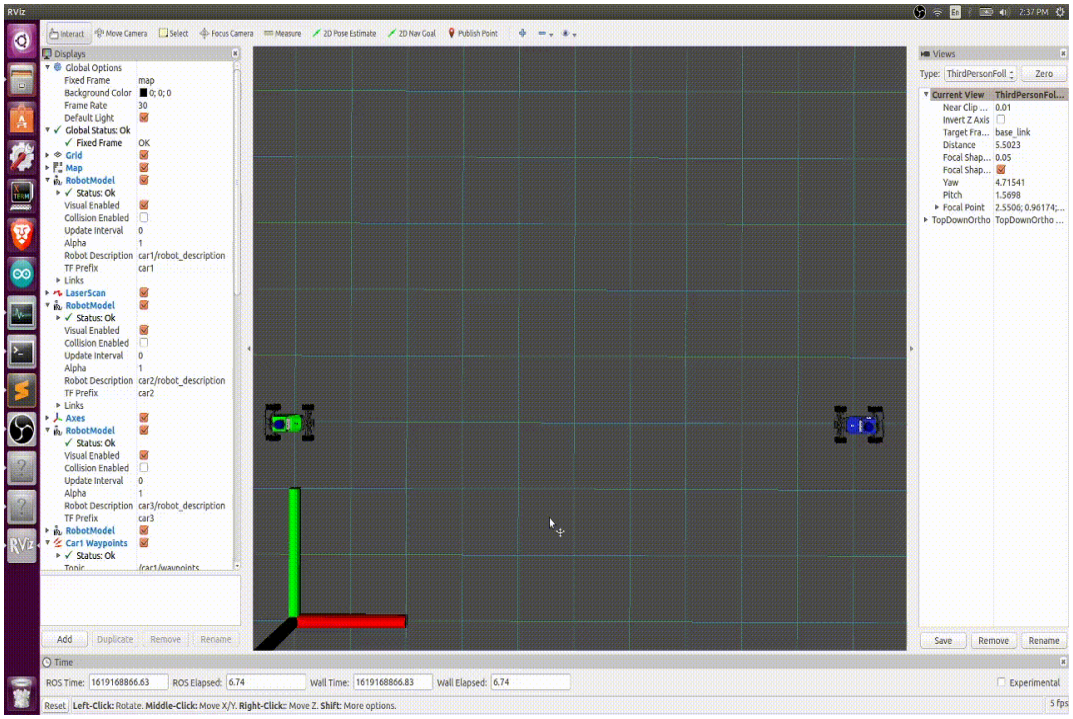
In this 

particular example, it may appear to be helping the agents. However, had there been another agent above/below these agents, it may have affected the performance adversely.

3) solver_time: The amount of time in milliseconds to be used for calculating the solution. It is not to be confused with the time to a collision. Increasing the solution time may lead to a better solution, however, as the system has to operate in real-time, this will reduce the control-loop frequency. A slower control-loop frequency results in a delayed response. A multi-agent navigation system responds to the actions of the agents around it. If the control loop is slower, the other agents may appear out of sync at times as they aren't running the control loop fast enough to respond to each other's actions.

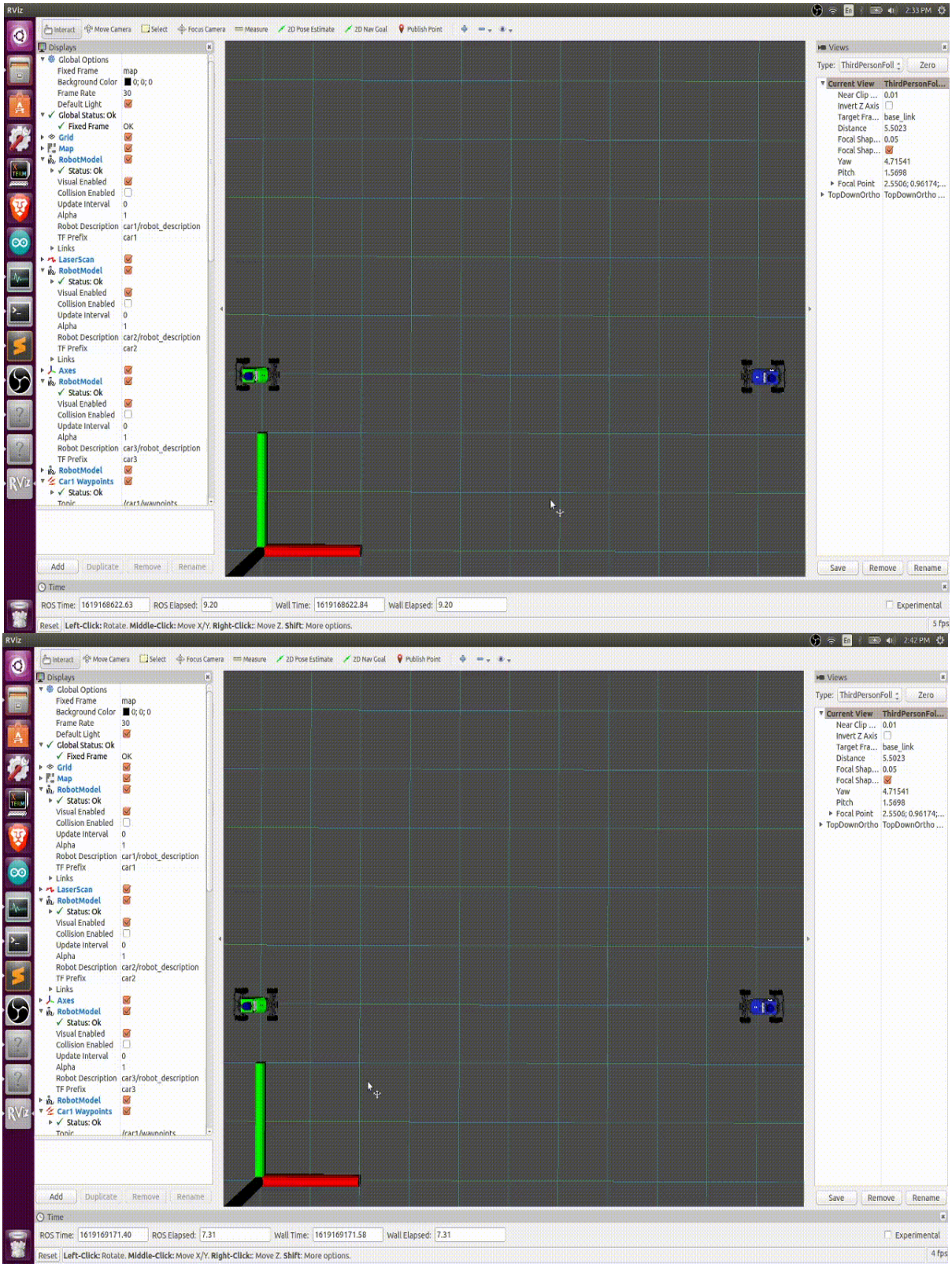
In multi-agent situations when the intent of the other agent isn't clear, all the agents are essentially guessing what they should do. If the first guess is wrong, it may lead to sub-optimal behavior. In the following figures, the solution time is 20 milliseconds for the first case and 30 milliseconds for the second. Notice how the blue car takes the wrong course of action initially in the second case.





At the same time, reducing the solution time may lead to the solution not converging at all, again, leading to sub-optimal solutions. Keeping the solution time between 15-30 milliseconds will yield feasible solutions, with larger times resulting in the above behavior appearing more often.

4) obey_time: The navigation system generally takes a waypoint array from a global planner. In multi-agent systems, the global planner will specify both space and time coordinates, as in, where the agent is supposed to be and when it is supposed to be there. If this parameter is set to false (as in the default demo), the car will disregard the timing. The first figure shows the behaviour for when this parameter is set to “false” and the second shows what happens when it is set to “true”:



As you can see, the blue car stops at the turn for a while before continuing. This happens because the blue car rounded the turn and arrived a little too early at the next waypoint. It therefore slowed down before continuing in order to maintain the timing.

Additional parameters like “allow_reverse”, “adaptive_lookahead” and “safety_radius” can be considered as “extra” or “experimental”. Allow_reverse sets whether the cars are allowed to go in reverse or not. It is set to true by default. Setting it to false may or may not cause issues with navigation. Adaptive_lookahead skips waypoints if they’re not reachable, this can help the system deal with global planners that provide unreasonable trajectories. The downside is that sometimes it may skip waypoints you do want to pass through. Last but not the least, the safety_radius is like an air-cushion around the car. A safety radius of 0 is not recommended, as localization errors in the real world may cause you to bump into other agents. For the MuSHR car, keep the safety radius as 0.1 + whatever the uncertainty of your localization system is.

6. Multi-Agent Coordination Planner for Multi-Goal Tasks

Plan and control multiple MuSHR cars to different goals without collisions using Enhanced Conflict-Based Search with Optimal Task Assignment (ECBS-TA) planning algorithm on a set of MuSHR cars. ECBS-TA produces a set of collision free trajectories, one for each car, from start to goal. By the end of the tutorial, a set of cars will be able to generate paths that avoid each other.

i. Enhanced Conflict-Based Search with Optimal Task Assignment (ECBS-TA)

Coordination Planner Problem Statement

There is a team of num_agents MuSHR cars and a set of num_goals tasks. Each task requires one car to follow all of the num_waypoints points and stop at the last one. The environment is a grid world where the movement of the car is restricted to orthongonal movement. This approximate approach can work well when combined with the nhttc controller. The coordination planner has the ability to create an initial path for each car that is avoiding all possible collisions. This path is leveraged with the nhttc controller to adapt a car’s path and avoid collisions in real-time if need be. The combination of these two services is more powerful than just the nhttc controller by itself as potential conflicts are sorted out prior to declaring each car’s path. It is possible that some cars will not have any tasks and others will have multiple to complete sequentially.

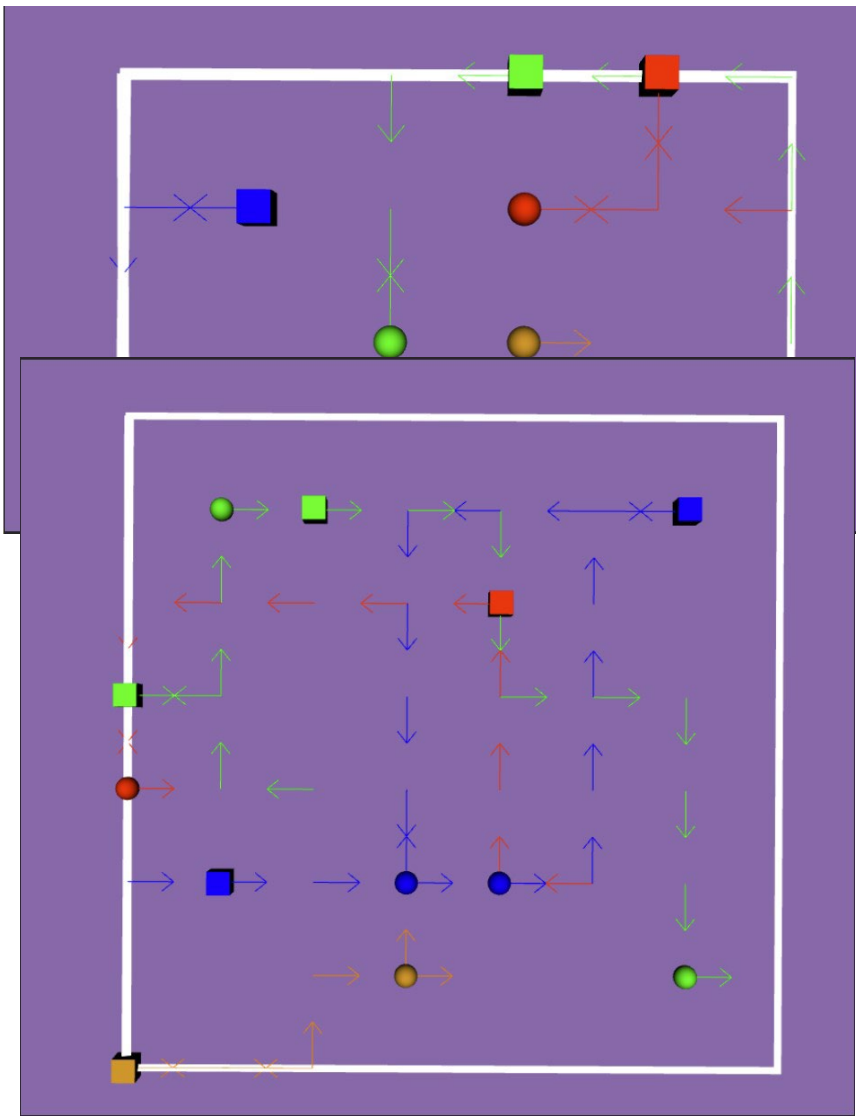
Examples of Initializing the Planner

Example #1

In this environment, there are 4 cars and have been given 4 tasks to complete within a 5 by 3 space. Each car get assigned to their respective task, in which their accumulated distance would be minimal.

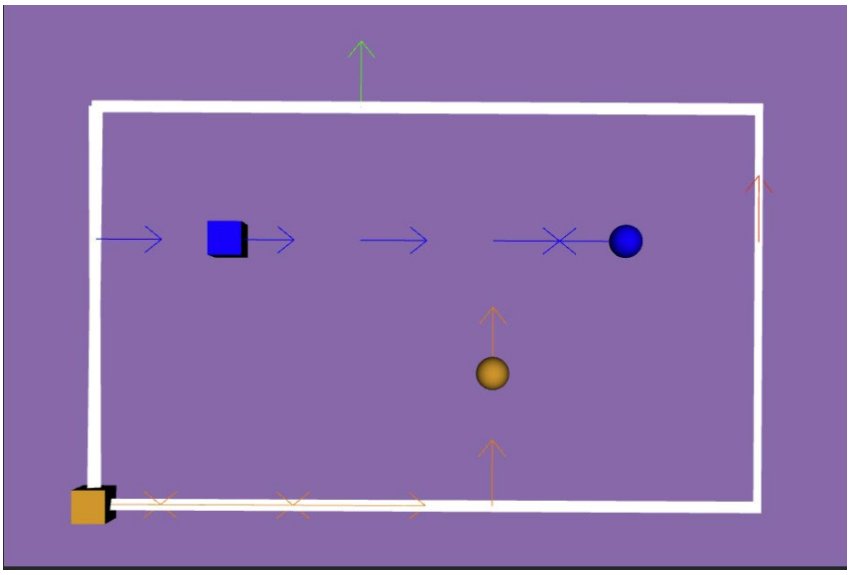
Example #2

In this environment, there are 4 cars and have been given 6 tasks to complete within a 6 by 6 space.
Number of tasks is more than number of cars, so the blue car and the green car have been given one task more than the others to complete.



Example #3

In this environment, there are 4 cars and have been given 2 tasks to complete within a 5 by 3 space. There is not enough task to assign to every cars. In this case, the red car and green car are idle and stay in place.



C. Behavioral Planner: High-level decision making

1. Concept

2. Options

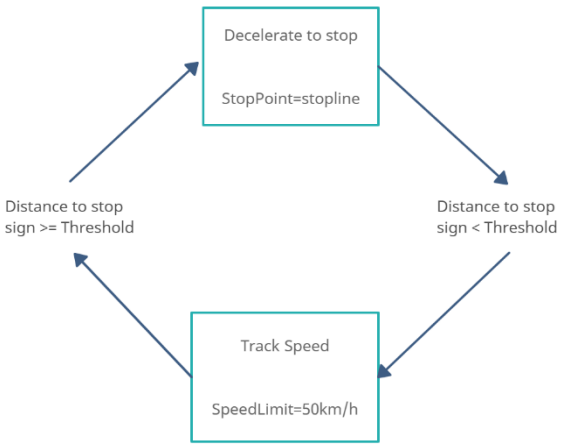
i. Finite State Machines (FSMs)

The first set of components of a finite state machine, is the set of states. For behavior planning system, the states will represent each of the possible driving maneuvers, which can be encountered.

In the example of handling a stop sign intersection with no traffic, we will only need two possible maneuvers or states, track speed and decelerate to stop. The maneuver decision defined by the behavior planner is set by the state of the finite state machine. Each state has associated with it an entry action, which is the action that is to be taken when a state is first entered.

For our behavior planner, these entry actions involve setting the necessary constraint outputs to accompany the behavior decision.

For instance, as soon as we enter the decelerate to stop state, we must also define the stopping point along the path. Similarly, the entry condition for the track speed state, sets the speed limit to track.



The second set of components of the finite state machine, are the transitions, which define the movement from one state to another. In our two-state example, we can transition from track speed to decelerate to stop, and from decelerate to stop back to track speed. Note that there can also be transitions which return us to the current state, triggering the entry action to repeat for that state. Each transition is accompanied with a set of transition conditions that need to be met before changing to the next state. These transition conditions are monitored while in a state to determine when a transition should occur. For our simple example, the transition conditions going from track speed to decelerate to stop involved checking if a stop point is within a threshold distance in our current lane.

Similarly, if we have reached zero velocity at the stop point, we can transition from decelerate to stop back to track speed. These two-state example highlights the most important aspects of the finite state machine-based behavior planner. As the number of scenarios and behaviors increases, the finite state machine that is needed becomes significantly more complex, with many more states and conditions for transition.

As a result of the decomposition of behavior planning into a set of states with transitions between them, the individual rules required remain relatively simple. This leads to straightforward implementations with clear divisions between separate behaviors. However, as the number of states increases, the complexity of defining all possible transitions and their conditions explodes. There is also no explicit way to handle uncertainty and errors in the input data. These challenges mean that the finite-state machine approach, as we approach full level 5 autonomy, tends to run into difficulties such as:

Rule-explosion: As we develop a more complete set of scenarios and maneuvers to handle, the number of rules required grows very quickly. This limitation means that while it is possible to develop a finite state machine behavior planner to handle a limited operational design domain, developing a full level 4 or 5 capable vehicle is almost impossible.

Noisy environment: While the addition of hyperparameters can be used to deal with some noise, this type of noise-handling is only able to deal with some very limited situations.

Hyperparameter tuning: As the behaviors required get more complex, the number of hyperparameters to both discretize the environment and handle some of the low-level noise grow rapidly. All these hyperparameters have to be tuned very carefully and this can be a lengthy process.

Unencountered situations: Due to the program nature of this approach, it is very likely that there will arise a situation in which the programmed logic of the system will react in an incorrect or unintended manner.

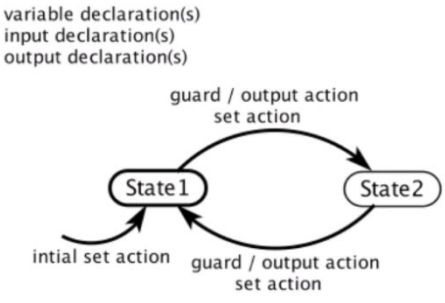
Maintainability: when adding or removing a state, it is necessary to change the conditions of all other states that have transition to the new or old one. Big changes are more susceptible to errors that may pass unnoticed.

Scalability: FSMs with many states lose the advantage of graphical readability, becoming a nightmare of boxes and arrows.

Reusability: as the conditions are inside the states, the coupling between the states is strong, being practically impossible to use the same behavior in multiple scenarios.

ii. Extended Finite State Machines (EFSMs)

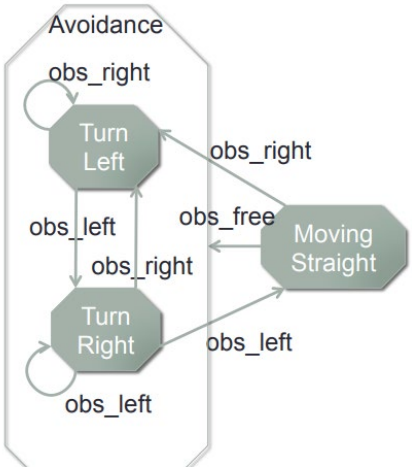
Extended finite-state machines (EFSM) are similar to conventional finite-state machines, but augmented with updates associated to the transitions; formulas constructed from variables, integer constants, the Boolean literals true and false, and the usual arithmetic and logic connectives. With an EFSM, we extend to state machines with variables that may be either read or written.



EFSMs suffer from the same problems as described for FSMs. Thus, we continued our research for alternatives.

iii. Hierarchical Finite State Machines (HFSMs): Harel's Statecharts

Harel's state machines are a mixture of mealy and Moore machines with three extra concepts:



Hierarchy: in this example of an obstacle avoidance, moving straight can each be thought of as part of a parent state. Organizing through hierarchy further compartmentalizes the design and can reduce the number of transition lines required to represent a system.

Orthogonality: the system might be comprised of multiple state machines operating simultaneously. While it's certainly possible to have two separate diagrams, in some cases, it may be convenient to display things together. We can illustrate this by including a Turn Right signal light in our state machine. The light turns on or off independent of the actions of the obstacle avoidance system.

Broadcasting: these mostly autonomous state machines could exchange information with one another. Let's say we want the Turn Right signal light to turn off every time we're in the Moving Straight state. So, what we will do is broadcast the light off event from the Moving Straight state, thus the light goes out when we want it to.

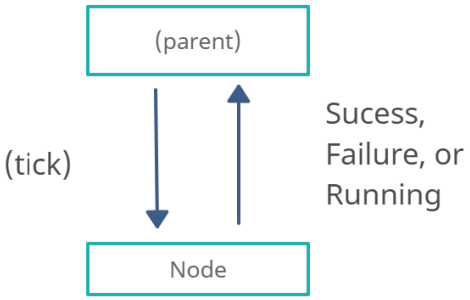
HFSM certainly provide a way to reuse transitions, but it's still not an ideal solution. The problem is that:

- Reusing transitions isn't trivial to achieve, and requires a lot of thought when you have to create logic for many different contexts.
- Editing transitions manually is rather tedious in the first place.

Another solution is to focus on making individual states modular so they can be easily reused as-is different parts of the logic. Behavior trees take this approach.

iv. Behavior Trees (BTs)

One of main problems with FSMs is that they're full of GOTO statements, which both harms modularity and increases complexity, making it very hard to debug in the case of complex problems. Another drawback of FSMs, is that the optimal transition structure is identical for all FSM-states. This is justified by the fact that the next action should not depend on the previous action, but on only on the current world state.



Behavior trees work by having a parent that "ticks" an action and then the action responds with Success, Failure, and Running. Thus, each action only needs to know if it succeeded, failed, or if its still trying. This creates a much more modular and weak dependence between subtrees. You don't need to know what actions to do next. This ticking and returning state are like a function call, instead of a GOTO statement.

	Finite State Machines (FSMs)	**Behavior Trees (BTs)**
Pros	Easy to implement	Modular: week dependence between subtrees
		Function call analogy (Instead of GOTO analogy)
	Easy to understand	
		Generalizes architectures: *Decision trees, Subsumption architecture, Teleo-reactive approach*
Cons	Does not scale well (n^2 possible transitions)	Harder to implement
	GOTO analogy problem (harms modularity)	
	State feedback is optimal *> Next action should only depend on current world state, not on current action*	
		Harder to understand
Verdict	=> Good for smaller problems	=> Good for larger problems

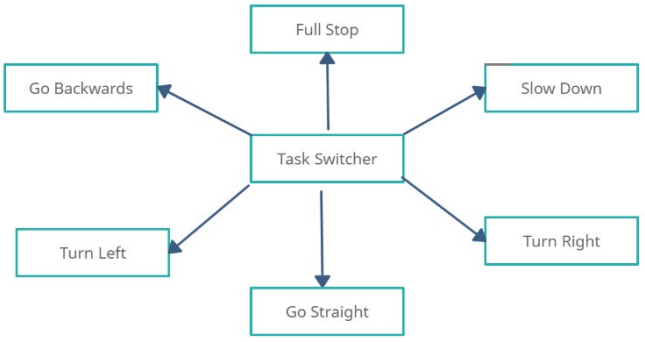
Other advantages of BTs include:

Maintainability: transitions in BT are defined by the structure, not by conditions inside the states. Because of this, nodes can be designed independent from each other, thus, when adding or removing new nodes (or even subtrees) in a small part of the tree, it is not necessary to change other parts of the model.

Scalability: when a BT have many nodes, it can be decomposed into small sub-trees saving the readability of the graphical model.

Reusability: due to the independence of nodes in BT, the subtrees are also independent. This allows the reuse of nodes or subtrees among other trees or projects.

The whole structure of a BT can be seen as a central task switcher, with all the actions at the leaves of the behavior tree, and interior nodes of a tree as a task switcher, deciding what to do next. And this task switcher is only depending on the world state, and not on an interior state of the tree.

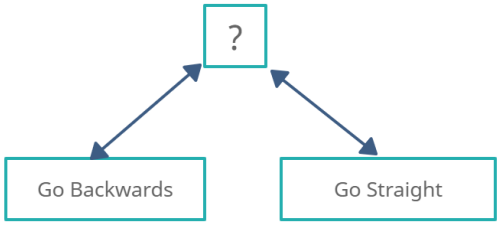


There are 2 fundamental compositions of actions that help to answer the question "What to do next?":

Fallback: Denoted by a question mark (?) and can be thought of as an OR function.

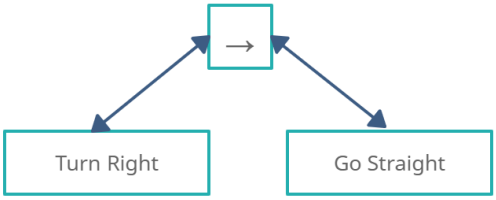
For example, let us consider the action of a simple parking scenario:

(Go Backwards ? Go Straight)



The rule is that we try to park by going backwards, and if we fail, then we try going straight (to adjust). But if "Go Backwards" succeeds, then we are done parking.

Sequence: Denoted by an arrow → and be thought of as an AND function.



For example, let us consider the action of changing lanes to the right. We first tick the "Turn Right" action, then if it succeeds, then we tick the "Go Straight" action.

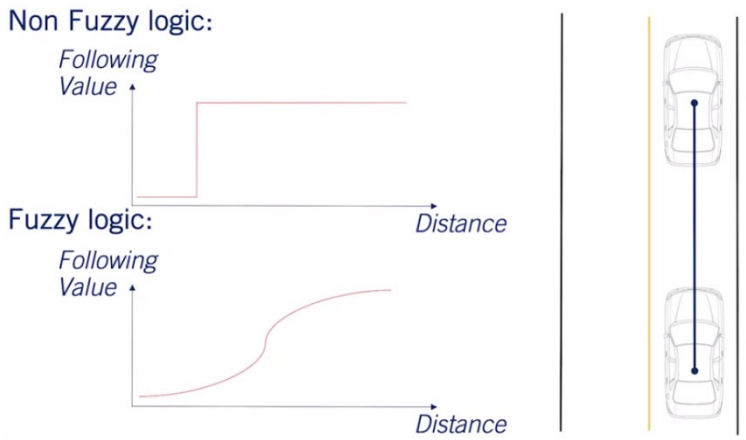
We can also connect these subtrees into a small behavior tree.

Behavior trees are run by a tick (Going down) that has a frequency dependent on the dynamics of the system.

Let us imagine a scenario where we tick the root (?), then tick “Go Backwards”, and this action fails, so the fallback takes the next child of the root which is “Go Straight”, imagine this returns Failure. So, the fallback takes the next child which is a sequence node, which itself takes its first child “Turn Right”, and if it succeeds, the sequence ticks “Go Straight”, and if this node returns Running, then this state goes all the way up to the OR root.

v. Fuzzy Logic

Fuzzy logic is a system by which a set of crisp, well-defined values are used to create a more continuous set of fuzzy states. For a simple example of how a Fuzzy based system works, let's take a look at the example of the vehicle following behavior.



Usually, we set a parametrized distance which divided the space into follow the vehicle or do not follow the vehicle. With a Fuzzy system we're able to have a continuous space over which different rules can be applied. For example, a Fuzzy system might react strongly to a lead vehicle when very close to it. But be less concerned with the speed matching or distance following requirements when it's further away.

While Fuzzy based rule systems are able to deal with the environmental noise of a system to a greater degree than traditional discreet systems, both rule-explosion and hyperparameter tuning remain issues with Fuzzy systems. In fact, Fuzzy systems can result in rule-explosion to an even greater degree, as even more logic is required to handle the fuzzy set of inputs. Accompanying the rule-explosion is a large challenge in hyperparameter tuning as well.

D. Local Re-Planner: Trajectory generation and optimization

1. Trajectory Generator

i. Concept

It decides how to construct the path that is compatible with the vehicle kinematics. Therefore, the aim of the path planning is to generate paths by taking the kinematic constraints of the vehicle into account to ensure the path feasibility.

ii. Reference Path

Straight line segments

Straight line segments are connected sequence of vertices. They could be useful if these vertices are equally spaced and provided in a straight line. Otherwise, the discontinuity of the line causes challenges for steered vehicle.

Waypoints

With a refinement of the line segment approach, waypoints can be constructed. The waypoints are tightly spaced points that has fixed spaces in distance and travel time. The relative position of the waypoints can be restricted to satisfy an approximate curvature constraint.

Parametrized curves

Parameterized curves can be constructed during motion planning. They provide a continuous varying motion and can be constructed to have smooth derivatives to help consistency of error and error rate calculations.

iii. Latice-based Path Planner

Resources

Trajectory Planning for an Indoor Mobile Robot Using Quintic Bezier Curves: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7418860&tag=1

Feasible Trajectories Generation for Autonomous Driving Vehicles: https://www.mdpi.com/2076-3417/11/23/11143

Time-Optimal Trajectory Planning Along Predefined Path for Mobile Robots with Velocity and Acceleration Constraints : https://lamor.fer.hr/images/50020777/Brezak2011.pdf

(Articulated robots) Lecture Note 9: Trajectory Generation: http://www2.ece.ohio-state.edu/~zhang/RoboticsClass/docs/LN9_TrajectoryGeneration.pdf

https://cdn.t3kys.com/media/upload/userFormUpload/bgUAt3tDP7kL71p1WzkFZ1Awfvv25MpQ.pdf

https://cdn.t3kys.com/media/upload/userFormUpload/ma6BpV6VMy6OvwRQoXtpNDBhysFeNUFp.pdf

---

As stated, a path can be generated as straight line segments, waypoints, and parametric curves. Parametric curves is the best option to choose because a curve is more similar to a real-life path generation. There are multiple options for parametric curves such as quintic splines and cubic spirals.

Position Quintic Polynomials

Quintic splines are fifth degree of polynomials. To generate x, y, θ, and K that are the position in an x and y dimension, the orientation and path curvature, respectively. The quintic splines provide a closed form solution advantage for the position. However, they evaluate the path curvature by taking derivative of the function of position/location. Because the derivative causes discontinuity along time, quintic splines are not preferred if the path curvature is important term for a process.

Cubic Polynomials

Cubic spirals are third order polynomials. They have a distinct closed form solution for the path curvature whereas it has no such solution for the location terms x and y. Although this case causes calculation challenges to evaluate the location terms, the cubic spirals are preferred to generate the parametric curve for the vehicle path. This is because, the path curvature which is a kinematic constraint, is more important to consider and the vehicle must not track a non-conformal path. The Cubic spiral terms and its equations are given below: (where s is the path length with respect to time)

$$\kappa(s) = a_3 s^3 + a_2 s^2 + a_1 s + a_0$$

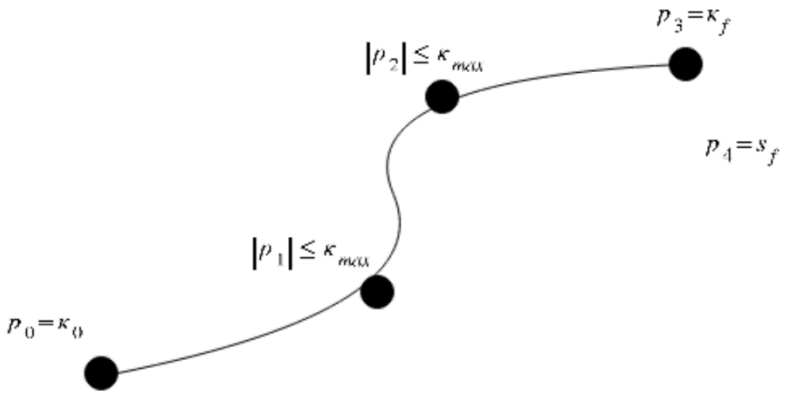
$$\theta(s) = \theta_0 + a_3 \frac{s^4}{4} + a_2 \frac{s^3}{3} + a_1 \frac{s^2}{2} + a_0 s$$

$$x(s) = x_0 + \int_0^s \cos(\theta(s'))\, ds'$$

$$y(s) = y_0 + \int_0^s \sin(\theta(s'))\, ds'$$

The position integrals are called Fresnel integrals and they need to be evaluated numerically. Simpson's rule is used to calculate the integrals. Simpson's rule is preferred because it is more accurate than the other methods such as midpoint and trapezoidal rules. The reason of accuracy comes from evaluating integrals of the quadratic interpolation of the given function rather than linear approaches.

Using Simpson's rule, the path curvatures on the points sf/3 and sf4 are evaluated where sf is the total length of the path. Figure 12 below shows the approximation of path curvatures



The path curvature is distributed using bending energy objective which is equal to the integral of square curvature along the path. It is represented with:

$$f_{be}(a_0, a_1, a_2, a_3, s_f)$$

Then, we add softening constraints to penalize deviation heavily in the objective function. Also, we assume that initial curvature is known, which correspond to a0. In the end, we have this function to be optimized such that:

$$min_{f_{be}(a_0, a_1, a_2, a_3, s_f) + \alpha(x_s(p_4) - x_f) + \beta(y_s(p_4) - y_f) + \gamma(\theta_s(p_4) - \theta_f)}$$

$$s.t.\ |p_1| \le \kappa_{max}\ and\ |p_2| \le \kappa_{max}$$

where α, β, γ are softening constraints; xf, yf, and θ f are the final locations and orientation, respectively.

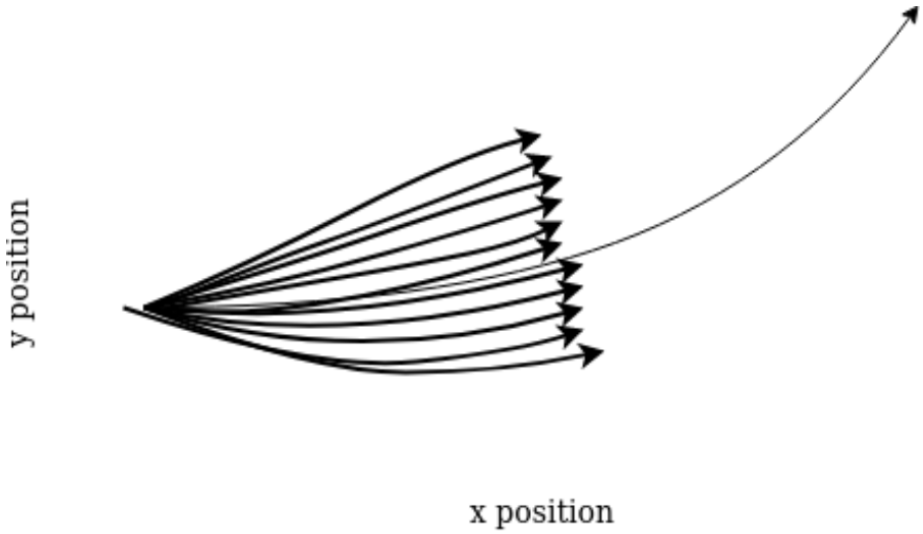
All the optimizations above are done by a single Python function in SciPy library. The minimization function evaluates the result by using 'L-BFGS-B' method and passing the required boundary.

A Conformal lattice planner has a goal horizon to generate the spirals. It determines a goal point in look ahead direction, then evaluates many points laterally offsetted with the first goal point, which is called goal set. Then, it generates the spirals due to the first and last point of the goal set.

After generating spirals to constraint the path curvature, the planner converts optimization variables back into the spiral parameters. This is because, the next step is sampling points along spiral using the spiral coefficients.

Sampling points is done by another numerical approximation method, that is trapezoidal rule integration. This evaluation is not proper to evaluate with Simpson's rule because of its hard calculation process. The trapezoidal method is more efficient because each subsequent point along the curve can be constructed from the previous one.

The Figure below shows the constructed curves. The number of curves will be a parameter in the code and it will be set to 1 to generate only one path.



Speed Quartic Polynomials

Symmetric Polynomials

iv. **-based Path Planner**

- **Straight Line Path**
- **CBB-RRT***

https://hal.archives-ouvertes.fr/hal-03188287/document

2. Trajectory Optimizer

i. Optimal Control Improvement

https://lamor.fer.hr/images/50020777/Brezak2011.pdf

3. Velocity Profile Generator

i. Concept

A vehicle must generate its velocity in a way that avoids collision with obstacles and other vehicles. If the leading vehicle velocity is smaller than our vehicle's velocity, then our vehicle must align its velocity with the leading's one. If the leading vehicle leaves from the look ahead distance, our vehicle can enlarge its velocity to the required one. Therefore, we need to have a velocity profile generator to manage such scenarios.

ii. Options

- **Trapezoidal Profile for Velocity Generation**

In path generation section, it is stated that path curvatures Ki are saved along the path. The vehicle velocity must not equal to an arbitrary value even if it is smaller than the leading vehicle velocity. This is because, the kinematic constraints of the vehicle have an effect on the vehicle's velocity. Using the curvature and the maximum lateral acceleration which is another kinematic constraint, the curvature velocity limit is evaluated by the given formula below:

$$v_k \leq \sqrt{\frac{a_{lat}}{\kappa_i}}$$

Then, the final velocity is selected as minimum of the curvature velocity limit, the behavior planner reference velocity, and the leading vehicle velocity as below:

$$v_f = min\left(v_{ref},\ v_{lead},\ v_k\right)$$

Now, the vehicle's initial velocity, final velocity, and maximum path length are known. Therefore, the acceleration of the vehicle can be evaluated. To ensure the acceleration does not exceed the boundary, it can be recalculated using formula below:

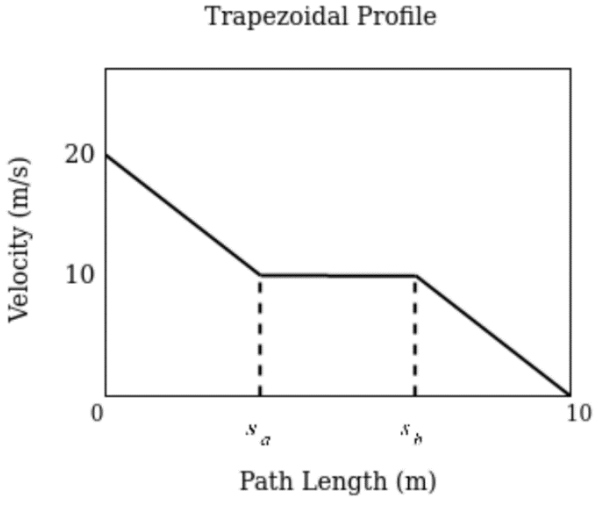
$$\frac{{v_f}^2 - {v_0}^2}{2s} = a \qquad \rightarrow \qquad \sqrt{2as + {v_0}^2} = v_f$$

To generate velocity profile, a trapezoidal profile generator is used because the velocity waits for a while when it ascends or descends to the final velocity rather than changing immediately linearly. An abstract trapezoidal profile is given with the related partial function below:

$$v_{f_i} = \sqrt{2a_0 s_i + {v_i}^2},\ s_i \leq s_a$$

$$v_{f_i} = v_t,\ s_a \leq s_i \leq s_b$$

$$v_{f_i} = \sqrt{2a_0\left(s_i - s_b\right) + {v_i}^2},\ s_b \leq s_i \leq s_f$$



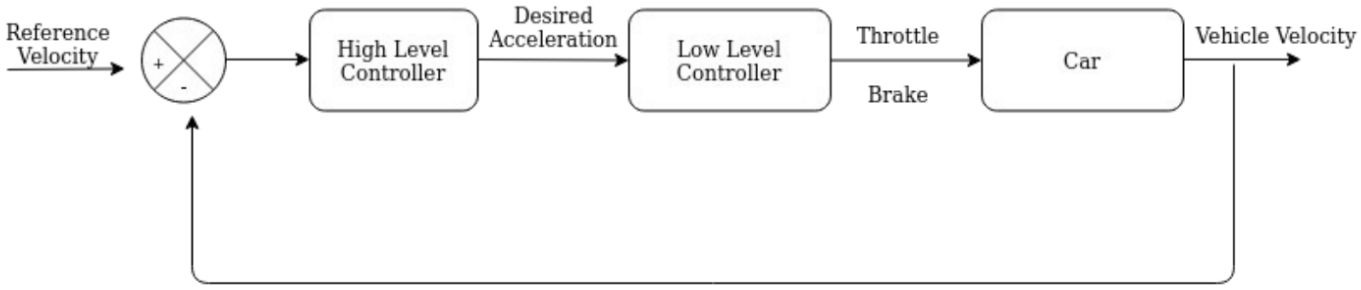
V. Motion Control

A. Longitudinal Control (Speed PID Controller)

1. Concept

Vehicle speed will be controlled by keeping at a reference speed by throttling and braking.

However, this longitudinal controller will handle just throttling, after completing the track, braking will be activated.



High Level Controller:

The high-level controller determines the desired acceleration for the vehicle based on the reference and the actual velocity where the PID gains are KP, KI and KD.

$$\ddot{X}des = K_P\left(\dot{X}ref - \dot{X}\right) + K_I\left(\int_0^t \left(\dot{X}ref - \dot{X}\right)dt\right) + K_D\left(\frac{d\left(\dot{X}ref - \dot{X}\right)}{dt}\right)$$

Low Level Controller:

The low-level controller will just convert the desired acceleration to throttle and brake by limiting the acceleration in a boundary and outputting as the throttle or brake in software.

2. Options

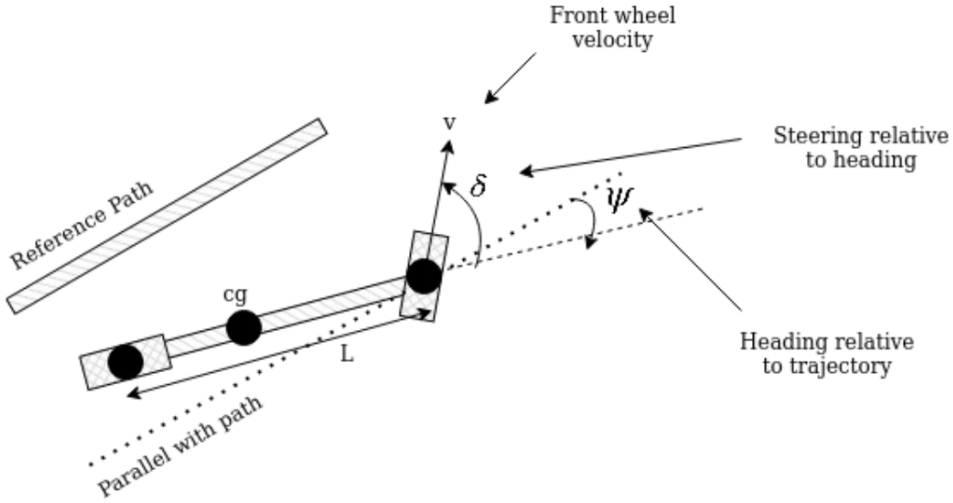
B. Lateral Control (Steering PID Controller)

1. Concept

To design a lateral controller for a robot, a reference path is needed to track.

Then, the lateral controller is responsible for making the robot follow the generated reference path with minimal errors. As such, there has to be an error term relative to the reference path. Then, a control law is required to drive the errors to zero and satisfy input constraints. Finally, adding dynamic considerations help the vehicle manage forces and moments acting on itself.

2. Controller Error Terms (Example: Bicycle Model)



There are two controller error terms: heading and cross track error.

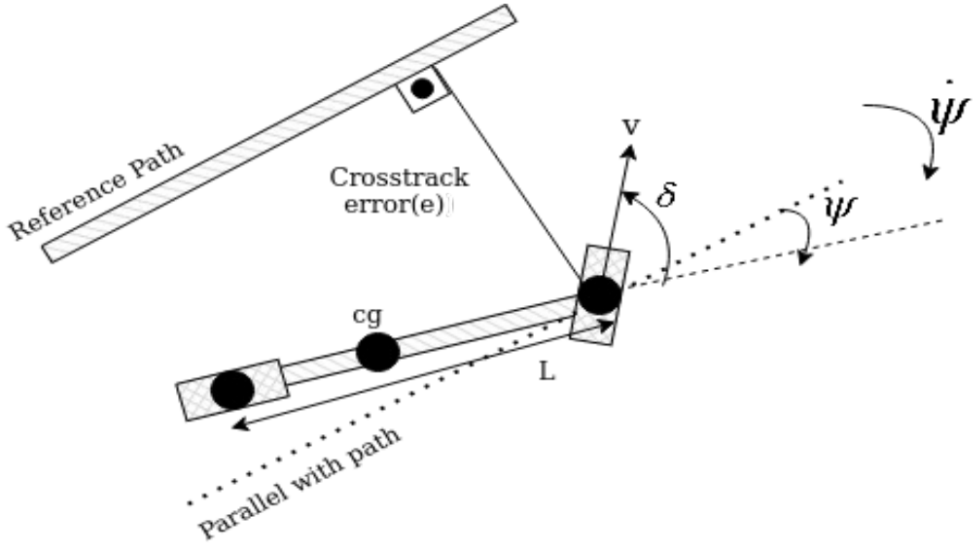
The heading error is the difference between path heading and vehicle heading at a reference point along the path. It states of how well aligned with the direction of the desired path. Then, the rate of heading error relative to the front axle is represented by the equation below

$$\dot{\psi}_{des}(t) - \psi(t) = \frac{v_f(t)\sin\delta(t)}{L}$$

For straight lines:

$$\dot{\psi}(t) = \frac{-v_f(t)\sin\delta(t)}{L}$$

The cross track error is the distance between the reference point on the vehicle and the closest point on the desired path. It states how close the vehicle position to the desired position along the path. Figure below shows the cross track error and the rate of change of heading error. For main purpose, both the heading and cross track errors must converge to zero.



The rate of change of the cross track error is defined equation below

$$\dot{e}(t) = v_f(t)\sin(\psi(t) - \delta(t))$$

As the velocity increases, the cross track error changes more quickly. It means smaller steering angle are need to correct for the same size cross track error.

3. Options

i. Geometric Path Tracking Solution: Stanley Approach

Any controller that uses only the geometry of the path and the vehicle kinematics to track a reference path is a geometric path tracking solution. These approaches ignore dynamic forces on the vehicles and assumes there are no slip holds at the wheels. Therefore, their performance suffer from slip conditions when the vehicle is aggressively maneuvered.

Stanley approach was firstly used by Stanford University's Darpa Grand Challenge Team. It uses the center of the front axle as a reference point and looks at both the heading and cross track errors. It defines a steering law by following up the process such that it:

• Corrects the heading error

• Corrects the cross track error

• Obeys maximum steering angle bounds

Let's introduce each step, then combine them. First, steering to align heading with desired heading eliminates heading error.

$$\delta(t) = \psi(t)$$

Then, steer to eliminate cross track error. This steering is proportional to cross track error and inversely proportional to velocity. An inverse tangent maps the proportional control signal to the angular range of -PI to PI for large errors. Gain k is determined experimentally. The formula given below shows the steering:

$$\delta(t) = \tan^{-1}\left(\frac{ke(t)}{v_f(t)}\right)$$

Finally, the controller uses a steering angle bound.

$$\delta(t) \in \left[\delta_{min}, \delta_{max}\right]$$

Combining the three steps into one line gives us Stanley control law:

$$\delta(t) = \psi(t) + \tan^{-1}\left(\frac{ke(t)}{v_f(t)}\right), \ \delta(t) \in \left[\delta_{min}, \delta_{max}\right]$$

When we look into the error dynamics of the law when the steering angle is not at the maximum:

$$\dot{e}(t) = -v_f(t)\sin(\psi(t) - \delta(t)) = -v_f(t)\sin\left(\tan^{-1}\left(\frac{ke(t)}{v_f(t)}\right)\right)$$

$$= \frac{-ke(t)}{\sqrt{1 + \left(\frac{ke(t)}{v_f}\right)^2}}$$

For small cross track errors the denominator can be simplified by assuming the quadratic term is negligible. Then, the equation turns out the ordinary differential equation such that

$$\dot{e}(t) \approx -ke(t)$$

A. Model Predictive Control

i. Receding Horizon Controller (RHC) Node

This node is responsible for motion planning and generating controls(steering, speed) for the car. The implementation shipped with the car uses a Model Predictive Controller (MPC) to generate control signals which are sent to the car's motor controller (VESC).

RHC is a model predictive controller that plans to waypoints from a goal (instead of a reference trajectory). This controller is suitable for cars that don't have a planning module, but want simple MPC.

librhc Layout

librhc (mushr_rhc_ros/src/librhc) is the core MPC code, with the other source being ROS interfacing code. The main components are:

- Cost function (librhc/cost): Takes into account the cost-to-go, collisions and other information to produce a cost for a set of trajectories.
- Model (librhc/model): A model of the car, currently using the kinematic bicycle model.
- Trajectory generation (librhc/trajgen): Strategies for generating trajectory libraries for MPC to evaluate.
- Value function (librhc/value): Evaluation of positions of the car with respect to a goal.
- World Representation (librhc/workrep): An occupancy grid based representation for the map.

mushr_rhc_ros ROS API

Publishers

Topic	Type	Description
/rhcontroller/markers	visualization_msgs/Marker	Halton points sampled in the map (for debugging purposes).
/rhcontroller/traj_chosen	geometry_msgs/PoseArray	The lowest cost trajectory (for debuggin purposes).
/car/mux/ackermann_cmd_mux/input/navigation	ackermann_msgs/AckermannDriveStamped	The lowest cost control to apply on the car.

Subscribers

Topic	Type	Description

/map_metadata	nav_msgs/MapMetaData	Uses dimension and resolution to create occupancy grid.
/move_base_simple/goal	geometry_msgs/PoseStamped	Goal to compute path to.
/car/car_pose	geometry_msgs/PoseStamped	*When using simulated car pose* Current pose of the car.
/car/pf/inferred_pose	geometry_msgs/PoseStamped	*When using particle filter for localization* Current pose of the car.

Services

Topic	Type	Description
/rhcontroller/reset/hard	std_srvs/Empty	Creates a new instance of the MPC object, redoing all initialization computation.
/rhcontroller/reset/soft	std_srvs/Empty	Resets parameters only, not redoing initialization.

B. State Estimation

1. Concept

2. Options

i. Particle Filter

MuSHR State Estimator

The Localization Node is implemented using a method called Particle Filtering which relies primarily on a data stream from the laser scanner.

mushr_pf: Particle Filter

Particle filter (pf) for localization using the laser scanner. The pf requires a map and laser scan.

Publishers

Topic	Type	Description
`/car/pf/inferred_pose`	geometry_msgs/PoseStamped	Particle filter pose estimate
`/car/pf/viz/particles`	geometry_msgs/PoseArray	Partilcle array. Good for debugging
`/car/pf/viz/laserpose`	geometry_msgs/PoseArray	Pose fo the laser

Subscribers

Topic	Type	Description
`/map`	nav_msgs/OccupancyGrid	Map the robot is in
`/car/scan`	sensor_msgs/LaserScan	Current laserscan
`/car/vesc/sensors/servo_position_command`	std_msgs/Float64	Current steering angle
`/car/vesc/sensors/core`	vesc_msgs/VescStateStamped	Current speed

ii. Kalman Filter

Extended Kalman Filter (EKF)

The state estimation of robots can be solved using an Extended Kalman filter (EKF). It is based on the linearization of the nonlinear maps (f, h) of around the estimated trajectory, and on the assumption.

$$
\begin{aligned}
&\hat{x}_{k+1|k} = f(\hat{x}_{k|k}, u_k) \\
&P_{k+1|k} = A_k P_{k|k} A_k' + W \\
&K_{k+1} = P_{k+1|k} C_{k+1}' (C_{k+1} P_{k+1|k} C_{k+1}' + V)^{-1} \\
&(\hat{\bar{s}}_k, \hat{\bar{c}}_k) = \mathbf{NBA}(\mathcal{I}_k, k) \\
&\hat{x}_{k+1|k+1} = \hat{x}_{k+1|k} + K_{k+1}(y_{k+1} - h(\hat{x}_{k+1|k}, (\hat{\bar{s}}_k, \hat{\bar{c}}_k))) \\
&P_{k+1|k+1} = P_{k+1|k} - K_{k+1} C_{k+1} P_{k+1|k}
\end{aligned}
$$

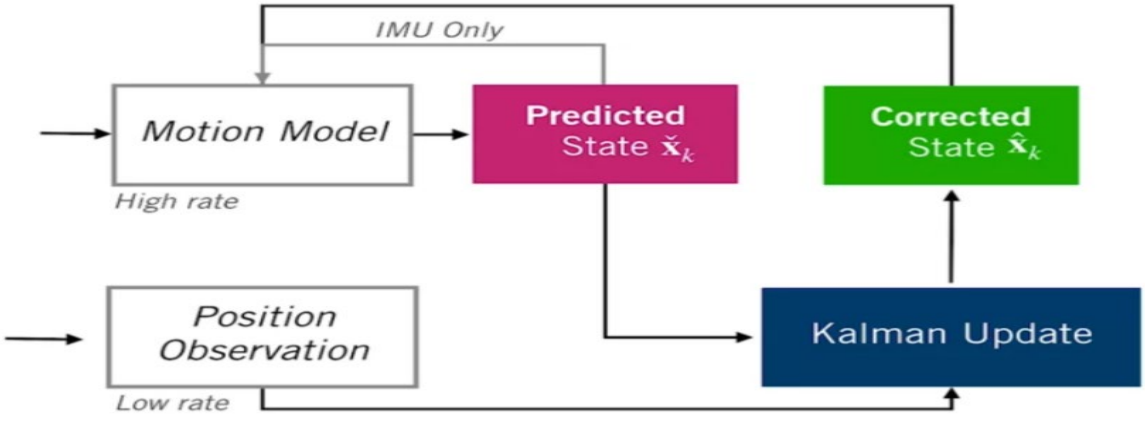
xˆk+1|k represents the estimate of xk+1 before getting the observation yk+1

xˆk+1|k+1 represents the estimate after getting that observation,

(ˆsk, ˆck) represent the output of NBA at time k and for each sensor Si , i ∈ Ik.

Error-State Extended Kalman Filter (ES-EKF)

The basic idea of Error State Extended Kalman Filter (ES-EKF) is presented in the following figure.



The state at each time step consists of position, velocity and orientation. The inputs of motion model are illustrated in following equations:

$$
\mathbf{x}_k = \begin{bmatrix} \mathbf{p}_k \\ \mathbf{v}_k \\ \mathbf{q}_k \end{bmatrix} \in R^{10} \qquad \mathbf{u}_k = \begin{bmatrix} \mathbf{f}_k \\ \boldsymbol{\omega}_k \end{bmatrix} \in R^6
$$

We present now the different equations for ES-EKF:

Error State

$$\delta \mathbf{x}_k = \begin{bmatrix} \delta \mathbf{p}_k \\ \delta \mathbf{v}_k \\ \delta \boldsymbol{\phi}_k \end{bmatrix} \in R^9$$

3x1 rotation error

Error Dynamics

$$\delta \mathbf{x}_k = \mathbf{F}_{k-1} \delta \mathbf{x}_{k-1} + \mathbf{L}_{k-1} \mathbf{n}_{k-1}$$

measurement noise

$$\mathbf{F}_{k-1} = \begin{bmatrix} \mathbf{1} & \mathbf{1}\Delta t & 0 \\ 0 & \mathbf{1} & -[\mathbf{C}_{ns} \mathbf{f}_{k-1}]_{\times} \Delta t \\ 0 & 0 & \mathbf{1} \end{bmatrix} \quad \mathbf{L}_{k-1} = \begin{bmatrix} 0 & 0 \\ \mathbf{1} & 0 \\ 0 & \mathbf{1} \end{bmatrix} \quad \mathbf{n}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_k)$$

Unscented Kalman Filter (UKF)

The Unscented Kalman Filter (UKF) has been implemented in recent years to overcome two main problems of the EKF. The UKF's principle is basically to find a transformation that allows to approximate and optimize the mean and covariance of a random vector of length n when it is transformed by a nonlinear map. The UKF estimate is accurate to the third order in the case of Gaussian noises.

Implementation of the UKF

Here we will describe the different steps of implementation of the UKF.

for each step k, starting from $\hat{x}_{k|k}$ and $P_{k|k}$, **do**

1) Compute $B_{k|k} = \sqrt{(n+\lambda)P_{k|k}}$, i.e., the scaled square root of matrix $P_{k|k}$
2) Compute the σ-points matrix

$$\chi_{k|k} = [\hat{x}_{k|k} \quad \hat{x}_{k|k} + B_{k|k} \quad \hat{x}_{k|k} - B_{k|k}] \in \mathbb{R}^{n \times (2n+1)}$$

There (and in the sequel) the sum of a vector plus a matrix is intended as summing the vector to all the column of the matrix (*à la* MATLAB)

3) Transform the σ-points matrix (columnwise)

$$\chi^*_{k+1|k} = f(\chi_{k|k}, u_k)$$

4) Compute the a-priori statistics

$$\hat{x}_{k+1|k} = \chi^*_{k+1|k} R^m$$
$$P_{k+1|k} = (\chi^*_{k+1|k} - \hat{x}_{k+1|k}) R^c (\chi^*_{k+1|k} - \hat{x}_{k+1|k})' + W$$

5) Update B and compute the new σ-points

$$B_{k+1|k} = \sqrt{(n+\lambda)P_{k+1|k}}$$
$$\chi_{k+1|k} = [\hat{x}_{k+1|k} \quad \hat{x}_{k+1|k} + B_{k+1|k} \quad \hat{x}_{k+1|k} - B_{k+1|k}]$$

6) Compute NBA

$$(\hat{\bar{s}}_k, \hat{\bar{c}}_k) = \mathbf{NBA}(\mathcal{I}_k, k)$$

7) Compute the predicted output

$$\Gamma_{k+1|k} = h(\chi_{k+1|k}, (\hat{\bar{s}}_k, \hat{\bar{c}}_k))$$
$$\hat{y}_{k+1|k} = \Gamma_{k+1|k} R^m$$

8) Compute the Kalman gain

$$P_{yy} = (\Gamma_{k+1|k} - \hat{y}_{k+1|k}) R^c (\Gamma_{k+1|k} - \hat{y}_{k+1|k})' + V$$
$$P_{xy} = (\chi_{k+1|k} - \hat{x}_{k+1|k}) R^c (\Gamma_{k+1|k} - \hat{y}_{k+1|k})'$$
$$K_{k+1} = P_{xy} P_{yy}^{-1}$$

9) Compute the a-posteriori statistics

$$\hat{x}_{k+1|k+1} = \hat{x}_{k+1|k} + K_{k+1}(y_{k+1} - \hat{y}_{k+1|k})$$
$$P_{k+1|k+1} = P_{k+1|k} - K_{k+1} P_{yy} K'_{k+1}$$

end

iii. Difference

In a linear system with Gaussian noise, the Kalman filter is optimal. In a system that is nonlinear, the Kalman filter can be used for state estimation, but the particle filter may give better results at the price of additional computational effort. In a system that has non-Gaussian noise, the Kalman filter is the optimal linear filter, but again the particle filter may perform better. The unscented Kalman filter (UKF) provides a balance between the low computational effort of the Kalman filter and the high performance of the particle filter.

The particle filter has some similarities with the UKF in that it transforms a set of points via known nonlinear equations and combines the results to estimate the mean and covariance of the state.

However, in the particle filter the points are chosen randomly, whereas in the UKF the points are chosen on the basis of a specific algorithm*. Because of this, the number of points used in a particle filter generally needs to be much greater than the number of points in a UKF. Another difference between the two filters is that the estimation error in a UKF does not converge to zero in any sense, but the estimation error in a particle filter does converge to zero as the number of particles (and hence the computational effort) approaches infinity.

*The unscented transformation is a method for calculating the statistics of a random variable which undergoes a nonlinear transformation and uses the intuition (which also applies to the particle filter) that it is easier to approximate a probability distribution than it is to approximate an arbitrary nonlinear function or transformation. See also this as an example of how the points are chosen in UKF."

VI. General Resources

Research and education in Robotics – Eurobot 2011,
https://link.springer.com/content/pdf/10.1007/978-3-642-21975-7.pdf