Unit 3: Set Up TEB Local Planner for a car

SUMMARY

Estimated time to completion: **1 hour**

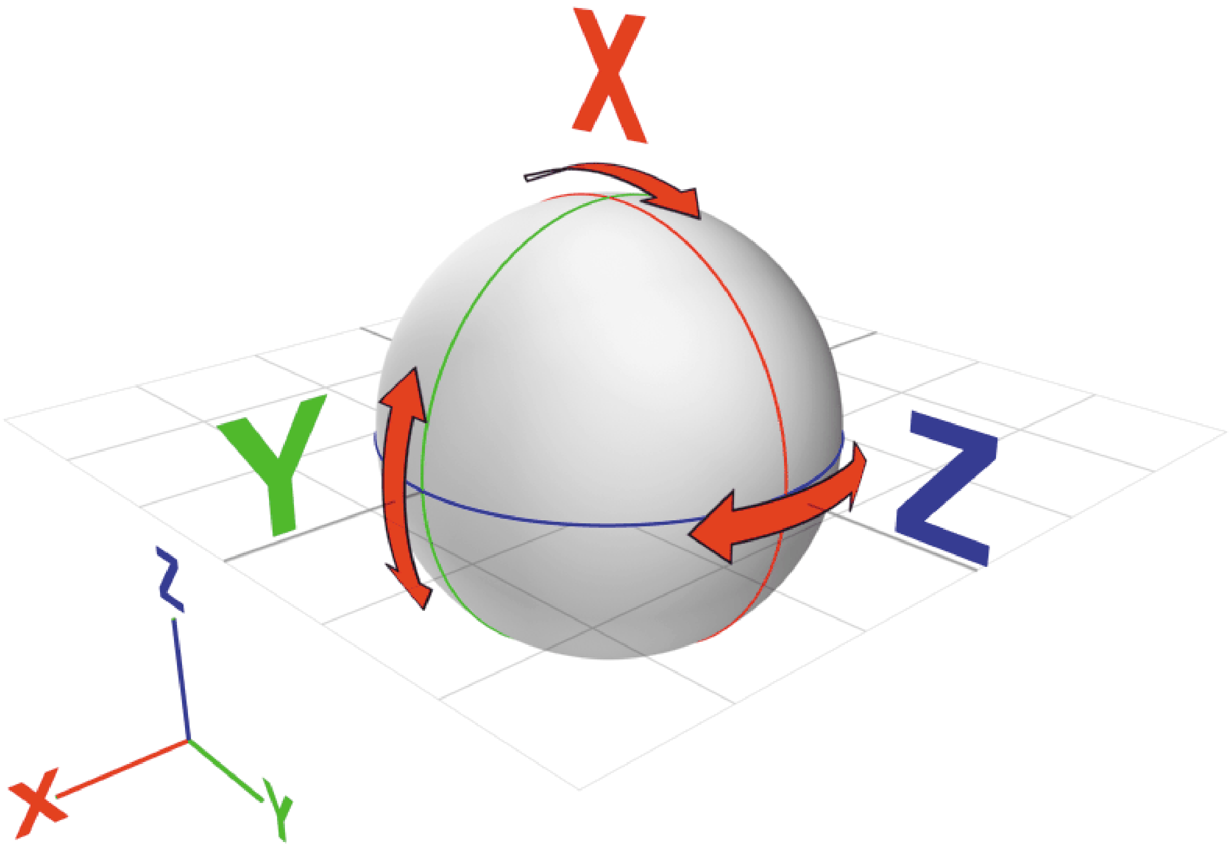
What will you learn with this unit?

- How does the TEB local planner perform Path Planning for car-like robots?
- What is the Footprint model?
- How to set up your environment for navigating with car-like robots.

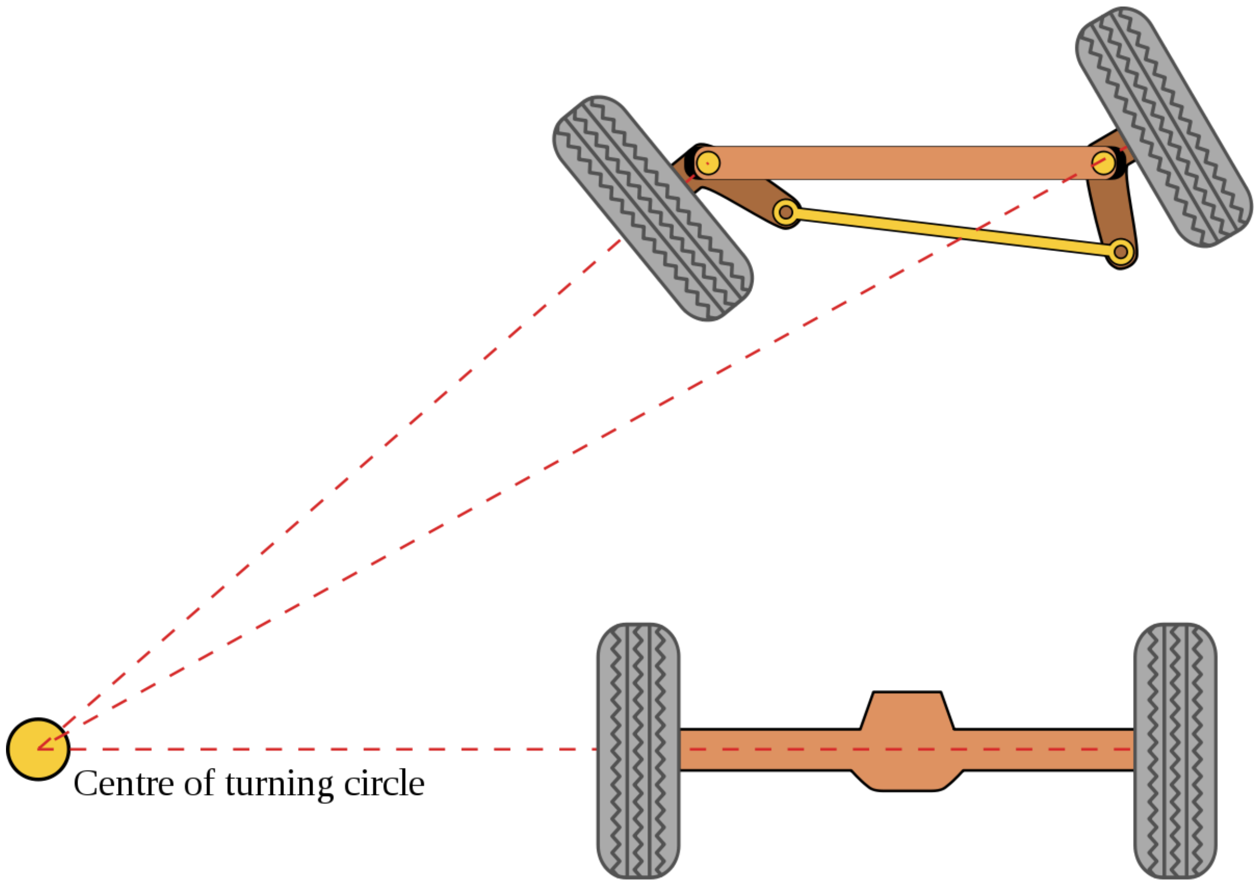
END OF SUMMARY

Up to this point, you've seen how to set up ROS Navigation for a robot, and how to set up and optimize the **teb_local_planner** for your robot. But now, let's go a step further.

By default, the ROS Navigation Stack is not meant to be used by car-like robots. In fact, it is meant to be used by robots that can rotate on their Z-axis.



However, the **teb_local_planner** is able to generate local plans that are feasible for Ackermann drives, as you saw in the demo for this course.



It does so by using a parameter that can set the minimum turning radius of the robot. This parameter is called **min_turning_radius**. The teb_local_planner must indeed adhere to the specifications of the Navigation Stack by providing a **geometry_msgs/Twist** message containing translational and angular velocities, which is the one used by the Navigation Stack, rather than providing an **ackermann_msgs/AckermannDriveStamped** message, which is the one usually used for commanding cars.
Thankfully, this simulation already handles all these conversions between Twist and Ackermann messages. But in case you need to do the conversions for your own simulation, you can have a look at the following code here, which converts Twist messages to Ackermann messages.
twist_to_ackermann_converter.py

In [ ]:

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

import rospy, math
from geometry_msgs.msg import Twist
from ackermann_msgs.msg import AckermannDriveStamped

def convert_trans_rot_vel_to_steering_angle(v, omega, wheelbase):
 if omega == 0 or v == 0:
   return 0
 radius = v / omega
 return math.atan(wheelbase / radius)


def cmd_callback(data):
 global wheelbase
 global ackermann_cmd_topic
 global frame_id
 global pub
  v = data.linear.x
 steering = convert_trans_rot_vel_to_steering_angle(v, data.angular.z,
wheelbase)
  msg = AckermannDriveStamped()
 msg.header.stamp = rospy.Time.now()
 msg.header.frame_id = frame_id
 msg.drive.steering_angle = steering
 msg.drive.speed = v
  pub.publish(msg)

if __name__ == '__main__':
 try:

   rospy.init_node('cmd_vel_to_ackermann_drive')

   twist_cmd_topic = rospy.get_param('~twist_cmd_topic', '/cmd_vel')
   ackermann_cmd_topic = rospy.get_param('~ackermann_cmd_topic',
'/ackermann_cmd')
   wheelbase = rospy.get_param('~wheelbase', 1.0)
   frame_id = rospy.get_param('~frame_id', 'odom')

   rospy.Subscriber(twist_cmd_topic, Twist, cmd_callback, queue_size=1)
   pub = rospy.Publisher(ackermann_cmd_topic, AckermannDriveStamped,
queue_size=1)

   rospy.loginfo("Node 'cmd_vel_to_ackermann_drive' started.\nListening to
%s, publishing to %s. Frame id: %s, wheelbase: %f", "/cmd_vel",
ackermann_cmd_topic, frame_id, wheelbase)

   rospy.spin()

 except rospy.ROSInterruptException:
   pass
```

Also, for car-like robots, it's required to be able to drive backwards, so the parameter **weight_kinematics_forward_drive** is ignored if the parameter **min_turning_radius** is non-zero.
Great! So, with all these introductions made, let's modify our current **teb_local_planners.yaml** file so that it now supports car-like robot navigation. But before that, we'll need to also modify some of the parameter files of the Navigation Stack, to take into account the new topics and frames of the simulation.

Exercise 3.1

Modify all the parameter files that need to be modified in order to support the new DBW MKZ simulation. For that, you'll need to have a look at the topics and frames that this new simulation uses, and modify them in all the parameter files where they are being used.
HINT: For instance, the **/odom** topic is now called **/catvehicle/odom**, and so is the frame associated with the odometry of the robot.

End of Exercise 3.1

Exercise 3.2

Now, modify the **teb_local_planner.yaml** file so that it supports car-like robots. This means, modifying the **min_turning_radius** parameter.

End of Exercise 3.2

Great! So, at this point, you're almost ready to support car-like robots with your Navigation setup. There's just one more thing you need to take into account: **the robot footprint model**.

Robot Footprint Model

Basically, the robot footprint model approximates the robot's 2D contour for optimization purposes. The model is crucial for the complexity of distance calculations and, hence, for the computation time. Therefore, the robot footprint model constitutes a dedicated parameter instead of loading the footprint from the common costmap_2d parameters. The optimization footprint model might differ from the costmap footprint model (which is instead used for the feasibility check).
The footprint model is selected and configured using the parameter server. Here you can see an example setup, including all the different types:

In [ ]:

```
TebLocalPlannerROS:
footprint_model: # types: "point", "circular", "line", "two_circles",
"polygon"
  type: "point"
  radius: 0.2 # for type "circular"
  line_start: [-0.3, 0.0] # for type "line"
  line_end: [0.3, 0.0] # for type "line"
  front_offset: 0.2 # for type "two_circles"
  front_radius: 0.2 # for type "two_circles"
  rear_offset: 0.2 # for type "two_circles"
  rear_radius: 0.2 # for type "two_circles"
  vertices: [ [0.25, -0.05], [0.18, -0.05], [0.18, -0.18], [-0.19, -0.18],
[-0.25, 0], [-0.19, 0.18], [0.18, 0.18], [0.18, 0.05], [0.25, 0.05] ] # for
type "polygon"
```

IMPORTANT NOTE: For car-like robots, the pose [0,0] is located at the rear-axle (axis of rotation).

There exist five different types of footprint models: point, circular, line, two circles, and polygon.

Point

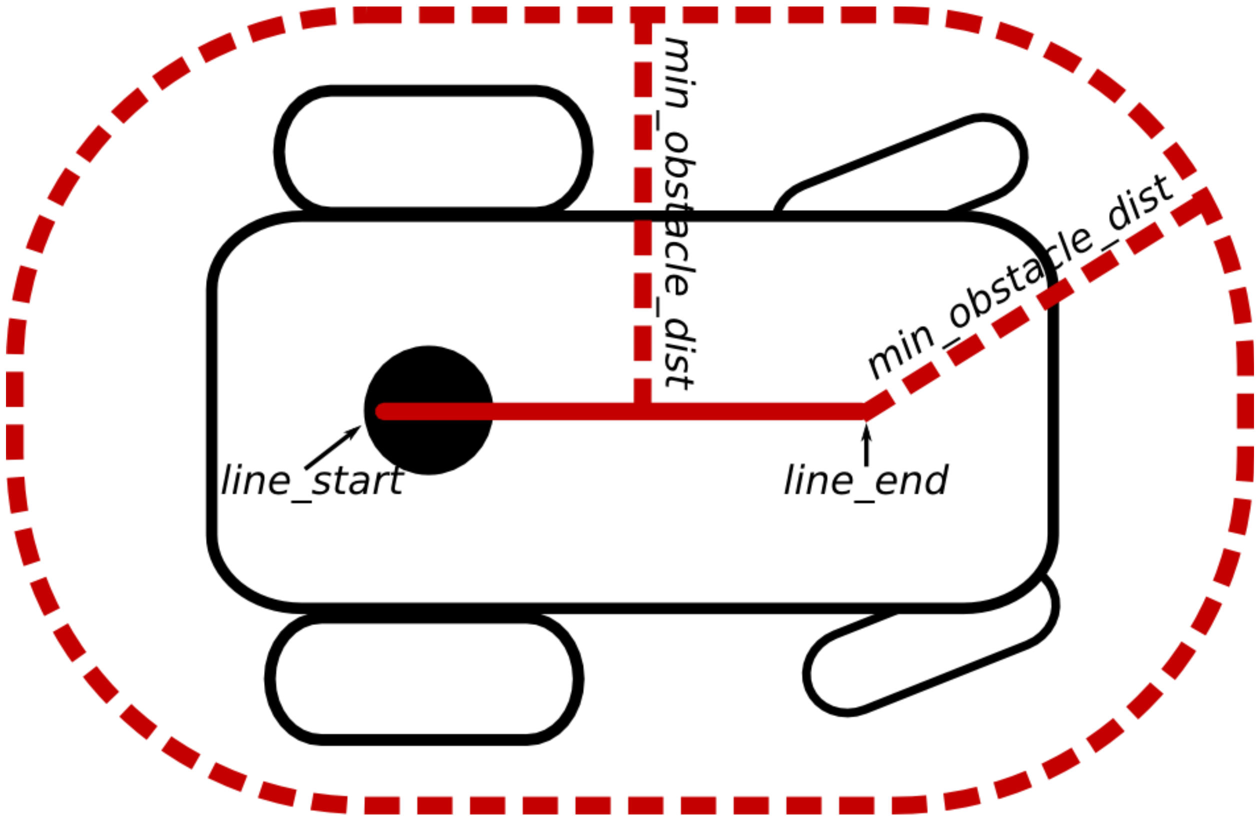
The robot is modeled as a single point. For this type, the least amount of computation time is required.

Circular

The robot is modeled as a simple circle with a given radius **~/footprint_model/radius**. The distance calculation is similar to that of the point-type robot, but with the exception that the robot's radius is added to the parameter **min_obstacle_dist** that each function calls. You can get rid of this extra addition by choosing a point-type robot and adding the radius to the minimum obstacle distance a-priori.

Line

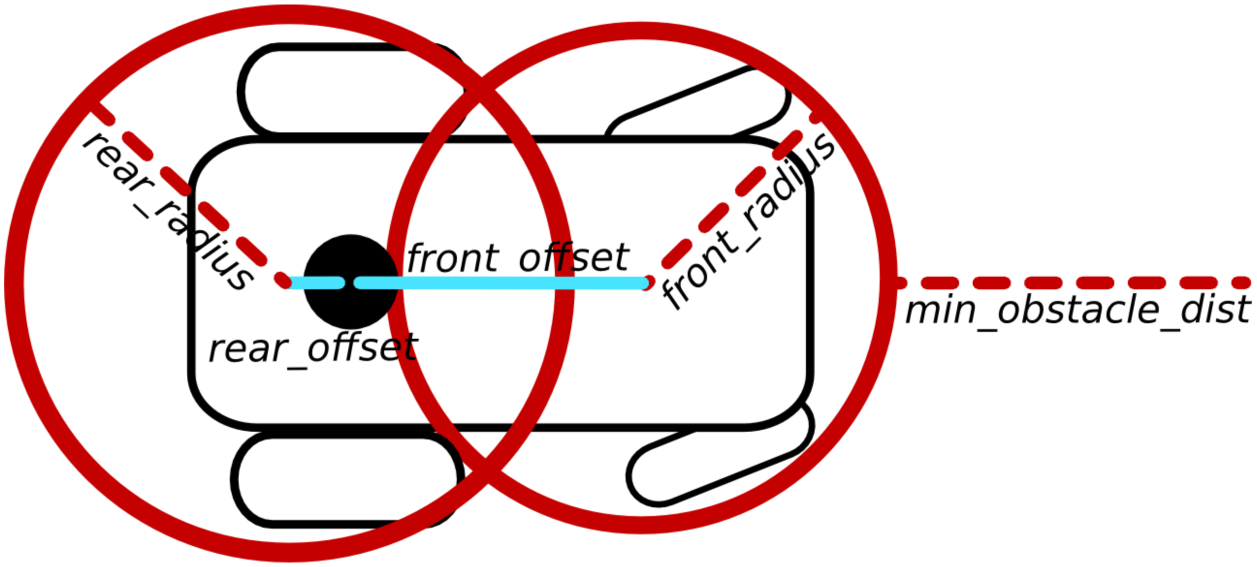
The line robot is useful for robots that exhibit different expansions/lengths in the longitudinal and lateral directions. The line (segment) can be configured using parameters **~/footprint_model/line_start** and **~/footprint_model/line_end** ([x,y] coordinates each). The robot (axis of rotation) is assumed to be at [0,0] (Unit: meters). Make sure to encapsulate the complete robot with further adjustment of the parameter **min_obstacle_dist** (see following image).



Two Circles

Another possibility to approximate the robot's contours consist of defining two circles. Each circle is described by an offset along the robot's x-axis and a radius: **~/footprint_model/front_offset**,

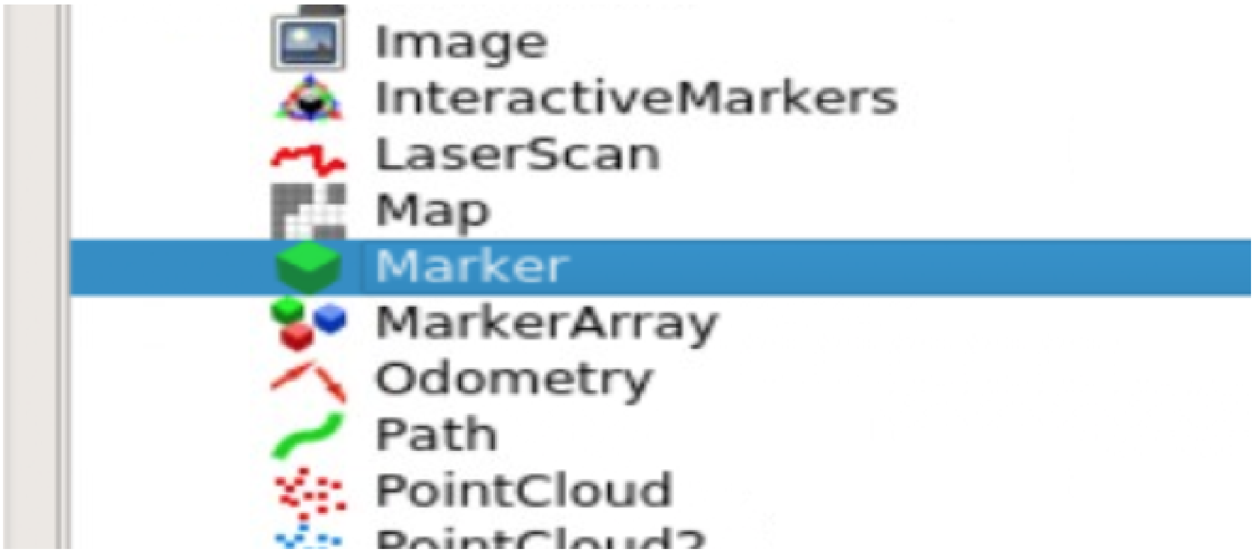
~/footprint_model/front_radius, **~/footprint_model/rear_offset**, and **~/footprint_model/rear_radius**. Offsets may be negative.



Polygon

A complex model can be incorporated by defining a closed polygon. The polygon is defined in terms of a list of vertices (provide x and y coordinates for each vertex). The robot's axis of rotation is assumed to be located at [0,0]. Do not repeat the first vertex since the polygon is closed automatically.

Keep in mind that each additional edge significantly increases the required computation time! You may copy your footprint model from your costmap common parameter file.
NOTE: The footprint model is published to ~/teb_markers, so it can be visualized through RVIZ by adding a Marker display.



NOTE 2: The footprint model won't be visible until you send a goal to the move_base node.

Exercise 3.3

Modify the **teb_local_planner.yaml** file so that it now adds the **footprint_model** parameter. Then, launch the move_base node and test how it performs.

End of Exercise 3.3

Exercise 3.4

Try to modify the **footprint_model** type, and see which one gives you better results.

End of Exercise 3.4

Exercise 3.5

Launch your setup with the **test_optim_node** and try to optimize your parameters.

End of Exercise 3.5

Congratulations!! You are now capable of navigating a car-like robot using ROS and the teb_local_planner!