AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

Problem 1: What does this command do? What parameters do you see listed?

This command lists all rosparameters which are in the current configuration.

rosparam list
/camera/image_raw/compressed/format
/camera/image_raw/compressed/jpeg_optimize
/camera/image_raw/compressed/jpeg_progressive
/camera/image_raw/compressed/jpeg_quality
/camera/image_raw/compressed/jpeg_restart_interval
/camera/image_raw/compressed/png_level
/camera/image_raw/compressedDepth/depth_max
/camera/image_raw/compressedDepth/depth_quantization
/camera/image_raw/compressedDepth/format
/camera/image_raw/compressedDepth/png_level
/camera/image_raw/theora/keyframe_frequency
/camera/image_raw/theora/optimize_for
/camera/image_raw/theora/quality
/camera/image_raw/theora/target_bitrate
/camera/imager_rate
/gazebo/auto_disable_bodies
/gazebo/cfm
/gazebo/contact_max_correcting_vel
/gazebo/contact_surface_layer
/gazebo/enable_ros_network
/gazebo/erp
/gazebo/gravity_x
/gazebo/gravity_y
/gazebo/gravity_z
/gazebo/max_contacts
/gazebo/max_update_rate
/gazebo/sor_pgs_iters
/gazebo/sor_pgs_precon_iters
/gazebo/sor_pgs_rms_error_tol
/gazebo/sor_pgs_w
/gazebo/time_step
/map
/navigator/k1
/navigator/k2

AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

/navigator/k3
/robot_description
/robot_state_publisher/publish_frequency
/rosdistro
/roslaunch/uris/host_genbu_stanford_edu__39347
/roslaunch/uris/host_genbu_stanford_edu__46081
/rosversion
/run_id
/rviz/compressed/mode
/sim
/turtlebot3_slam_gmapping/angularUpdate
/turtlebot3_slam_gmapping/astep
/turtlebot3_slam_gmapping/base_frame
/turtlebot3_slam_gmapping/delta
/turtlebot3_slam_gmapping/iterations
/turtlebot3_slam_gmapping/kernelSize
/turtlebot3_slam_gmapping/lasamplerange
/turtlebot3_slam_gmapping/lasamplestep
/turtlebot3_slam_gmapping/linearUpdate
/turtlebot3_slam_gmapping/llsamplerange
/turtlebot3_slam_gmapping/llsamplestep
/turtlebot3_slam_gmapping/lsigma
/turtlebot3_slam_gmapping/lskip
/turtlebot3_slam_gmapping/lstep
/turtlebot3_slam_gmapping/map_update_interval
/turtlebot3_slam_gmapping/maxUrange
/turtlebot3_slam_gmapping/minimumScore
/turtlebot3_slam_gmapping/odom_frame
/turtlebot3_slam_gmapping/ogain
/turtlebot3_slam_gmapping/particles
/turtlebot3_slam_gmapping/resampleThreshold
/turtlebot3_slam_gmapping/sigma
/turtlebot3_slam_gmapping/srr
/turtlebot3_slam_gmapping/srt
/turtlebot3_slam_gmapping/str
/turtlebot3_slam_gmapping/stt
/turtlebot3_slam_gmapping/temporalUpdate
/turtlebot3_slam_gmapping/xmax
/turtlebot3_slam_gmapping/xmin
/turtlebot3_slam_gmapping/ymax
/turtlebot3_slam_gmapping/ymin
/use_sim_time

AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

Problem 2: Where exactly are those being set? Start from section7.launch and trace back through all of the launch files that are called as a result. For each launch file, list a few rosparams that are set within the file or state that none are set in the file.

Root.launch
- Rosparam for sim, map, rviz
- Empty_world.launch
- Gmapping_config.launch
    - Creating the namespace variables for turtlebot3_slam_gmapping
- Goal_commander.py
- navigator.py

Problem 3: Try using rosparam get to get the values of 2-3 parameters. List the parameter names and their values.sim

sim: true
map: true
turtlebot3_slam_gmapping/angularUpdate: 0.2

Problem 4: Include your code. Modify the launch of navigator node inside section7.launch

```
<launch>
  <arg name="sim" default="true"/>

  <include file="$(find asl_turtlebot)/launch/root.launch">
    <arg name="world" value="project_city" />
    <arg name="x_pos" default="3.15"/>
    <arg name="y_pos" default="1.6"/>
    <arg name="z_pos" default="0.0"/>
    <arg name="rviz" default="section4"/>
    <arg name="model" default="asl_turtlebot"/>
    <arg name="sim" default="$(arg sim)"/>
  </include>
```

AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

```
  <node pkg="asl_turtlebot" type="navigator.py" name="navigator" output="screen" >
    <param name="v_max" value="0.2"/>
    <param name="om_max" value="0.4"/>
  </node>
</launch>
```

Problem 5: Include your code. Modify navigator.py to get the value of these parameters from the ROS param server

```
self.v_max =  rospy.get_param("navigator/v_max") # maximum velocity
self.om_max = rospy.get_param("navigator/om_max")  # maximum angular velocity
```

Problem 6: Test these launch files with your robot sim and paste the contents of your section7.launch and section7_slow.launch files into your submission.

Section7_slow.launch

```
<launch>
  <arg name="sim" default="true"/>

  <include file="$(find asl_turtlebot)/launch/sections/section7.launch">
    <arg name="v_max" value="0.1" />
    <arg name="om_max" value="0.2"/>
    <arg name="sim" value="$(arg sim)"/>
  </include>



</launch>
```

Section7.launch

```
<launch>
  <arg name="sim" default="true"/>
  <arg name="v_max" default="0.2" />
  <arg name="om_max" default="0.4" />

  <include file="$(find asl_turtlebot)/launch/root.launch">
```

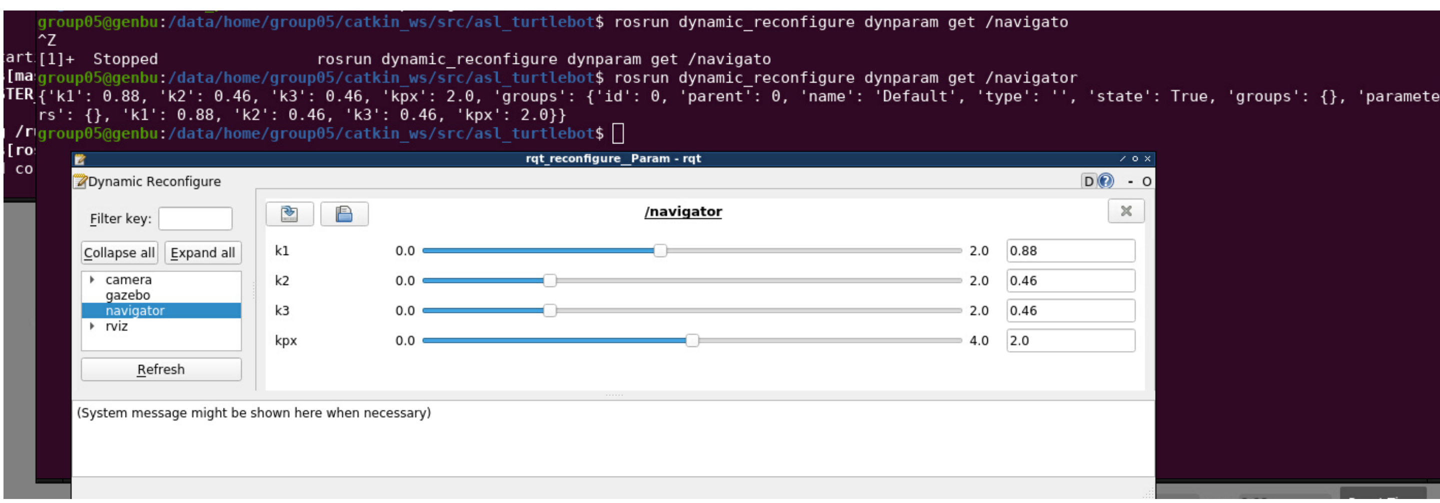
AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

<arg name="world" value="project_city" />
<arg name="x_pos" default="3.15"/>
<arg name="y_pos" default="1.6"/>
<arg name="z_pos" default="0.0"/>
<arg name="rviz" default="section4"/>
<arg name="model" default="asl_turtlebot"/>
<arg name="sim" default="$(arg sim)"/>
</include>

<node pkg="asl_turtlebot" type="navigator.py" name="navigator" output="screen">
<param name="v_max" value="$(arg v_max)"/>
<param name="om_max" value="$(arg om_max)"/>
</node>
</launch>

Problem 7: Test this on your robot sim and paste the contents of your navigator.cfg and navigator.py files into your submission.



#navigator.cfg

#!/usr/bin/env python3
PACKAGE = "asl_turtlebot"

from dynamic_reconfigure.parameter_generator_catkin import *

AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

```
gen = ParameterGenerator()

gen.add("k1",       double_t,    0, "Pose Controller k1", 0.8,  0., 2.0)
gen.add("k2",       double_t,    0, "Pose Controller k2", 0.4,  0., 2.0)
gen.add("k3",       double_t,    0, "Pose Controller k3", 0.4,  0., 2.0)
gen.add("kpx",       double_t,    0, "Pose Controller kpx", 2,  0., 4.0)

exit(gen.generate(PACKAGE, "navigator", "Navigator"))


#navigator.py

#!/usr/bin/env python3

import rospy
from nav_msgs.msg import OccupancyGrid, MapMetaData, Path
from geometry_msgs.msg import Twist, Pose2D, PoseStamped
from std_msgs.msg import String
import tf
import numpy as np
from numpy import linalg
from utils.utils import wrapToPi
from utils.grids import StochOccupancyGrid2D
from planners import AStar, compute_smoothed_traj
import scipy.interpolate
import matplotlib.pyplot as plt
from controllers import PoseController, TrajectoryTracker, HeadingController
from enum import Enum

from dynamic_reconfigure.server import Server
from asl_turtlebot.cfg import NavigatorConfig

# state machine modes, not all implemented
class Mode(Enum):
    IDLE = 0
    ALIGN = 1
    TRACK = 2
    PARK = 3


class Navigator:
```

AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

```
"""
This node handles point to point turtlebot motion, avoiding obstacles.
It is the sole node that should publish to cmd_vel
"""

def __init__(self):
    rospy.init_node("turtlebot_navigator", anonymous=True)
    self.mode = Mode.IDLE

    # current state
    self.x = 0.0
    self.y = 0.0
    self.theta = 0.0

    # goal state
    self.x_g = None
    self.y_g = None
    self.theta_g = None

    self.th_init = 0.0

    # map parameters
    self.map_width = 0
    self.map_height = 0
    self.map_resolution = 0
    self.map_origin = [0, 0]
    self.map_probs = []
    self.occupancy = None
    self.occupancy_updated = False

    # plan parameters
    self.plan_resolution = 0.1
    self.plan_horizon = 15

    # time when we started following the plan
    self.current_plan_start_time = rospy.get_rostime()
    self.current_plan_duration = 0
    self.plan_start = [0.0, 0.0]

    # Robot limits
    self.v_max =  rospy.get_param("navigator/v_max") # maximum velocity
    self.om_max = rospy.get_param("navigator/om_max")  # maximum angular velocity
```

AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

```
        self.v_des = 0.12  # desired cruising velocity
        self.theta_start_thresh = 0.05  # threshold in theta to start moving forward when
path-following
        self.start_pos_thresh = (
            0.2  # threshold to be far enough into the plan to recompute it
        )

        # threshold at which navigator switches from trajectory to pose control
        self.near_thresh = 0.2
        self.at_thresh = 0.02
        self.at_thresh_theta = 0.05

        # trajectory smoothing
        self.spline_alpha = 0.15
        self.spline_deg = 3  # cubic spline
        self.traj_dt = 0.1

        # trajectory tracking controller parameters
        self.kpx = 2
        self.kpy = 1.0
        self.kdx = 1.5
        self.kdy = 1.5

        # heading controller parameters
        self.kp_th = 2.0

        self.traj_controller = TrajectoryTracker(
            self.kpx, self.kpy, self.kdx, self.kdy, self.v_max, self.om_max
        )
        self.pose_controller = PoseController(
            0.0, 0.0, 0.0, self.v_max, self.om_max
        )
        self.heading_controller = HeadingController(self.kp_th, self.om_max)

        self.nav_planned_path_pub = rospy.Publisher(
            "/planned_path", Path, queue_size=10
        )
        self.nav_smoothed_path_pub = rospy.Publisher(
            "/cmd_smoothed_path", Path, queue_size=10
        )
        self.nav_smoothed_path_rej_pub = rospy.Publisher(
```

AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

```
        "/cmd_smoothed_path_rejected", Path, queue_size=10
    )
    self.nav_vel_pub = rospy.Publisher("/cmd_vel", Twist, queue_size=10)

    self.trans_listener = tf.TransformListener()

    self.cfg_srv = Server(NavigatorConfig, self.dyn_cfg_callback)

    rospy.Subscriber("/map", OccupancyGrid, self.map_callback)
    rospy.Subscriber("/map_metadata", MapMetaData, self.map_md_callback)
    rospy.Subscriber("/cmd_nav", Pose2D, self.cmd_nav_callback)

    print("finished init")

def dyn_cfg_callback(self, config, level):
    rospy.loginfo(
        "Reconfigure Request: k1:{k1}, k2:{k2}, k3:{k3}, kpx:{kpx}".format(**config)
    )
    self.pose_controller.k1 = config["k1"]
    self.pose_controller.k2 = config["k2"]
    self.pose_controller.k3 = config["k3"]
    self.kpx = config["kpx"]
    return config

def cmd_nav_callback(self, data):
    """
    loads in goal if different from current goal, and replans
    """
    if (
        data.x != self.x_g
        or data.y != self.y_g
        or data.theta != self.theta_g
    ):
        rospy.logdebug(f"New command nav received:\n{data}")
        self.x_g = data.x
        self.y_g = data.y
        self.theta_g = data.theta
        self.replan()

def map_md_callback(self, msg):
    """
    receives maps meta data and stores it
```

AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

```
    """
    self.map_width = msg.width
    self.map_height = msg.height
    self.map_resolution = msg.resolution
    self.map_origin = (msg.origin.position.x, msg.origin.position.y)

def map_callback(self, msg):
    """
    receives new map info and updates the map
    """
    self.map_probs = msg.data
    # if we've received the map metadata and have a way to update it:
    if (
        self.map_width > 0
        and self.map_height > 0
        and len(self.map_probs) > 0
    ):
        self.occupancy = StochOccupancyGrid2D(
            self.map_resolution,
            self.map_width,
            self.map_height,
            self.map_origin[0],
            self.map_origin[1],
            7,
            self.map_probs,
        )
        if self.x_g is not None:
            # if we have a goal to plan to, replan
            rospy.loginfo("replanning because of new map")
            self.replan()  # new map, need to replan

def shutdown_callback(self):
    """
    publishes zero velocities upon rospy shutdown
    """
    cmd_vel = Twist()
    cmd_vel.linear.x = 0.0
    cmd_vel.angular.z = 0.0
    self.nav_vel_pub.publish(cmd_vel)

def near_goal(self):
    """
```

AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

```
    returns whether the robot is close enough in position to the goal to
    start using the pose controller
    """
    return (
        linalg.norm(np.array([self.x - self.x_g, self.y - self.y_g]))
        < self.near_thresh
    )

def at_goal(self):
    """
    returns whether the robot has reached the goal position with enough
    accuracy to return to idle state
    """
    return (
        linalg.norm(np.array([self.x - self.x_g, self.y - self.y_g]))
        < self.at_thresh
        and abs(wrapToPi(self.theta - self.theta_g)) < self.at_thresh_theta
    )

def aligned(self):
    """
    returns whether robot is aligned with starting direction of path
    (enough to switch to tracking controller)
    """
    return (
        abs(wrapToPi(self.theta - self.th_init)) < self.theta_start_thresh
    )

def close_to_plan_start(self):
    return (
        abs(self.x - self.plan_start[0]) < self.start_pos_thresh
        and abs(self.y - self.plan_start[1]) < self.start_pos_thresh
    )

def snap_to_grid(self, x):
    return (
        self.plan_resolution * round(x[0] / self.plan_resolution),
        self.plan_resolution * round(x[1] / self.plan_resolution),
    )

def switch_mode(self, new_mode):
    rospy.loginfo("Switching from %s -> %s", self.mode, new_mode)
```

AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

```
    self.mode = new_mode

def publish_planned_path(self, path, publisher):
    # publish planned plan for visualization
    path_msg = Path()
    path_msg.header.frame_id = "map"
    for state in path:
        pose_st = PoseStamped()
        pose_st.pose.position.x = state[0]
        pose_st.pose.position.y = state[1]
        pose_st.pose.orientation.w = 1
        pose_st.header.frame_id = "map"
        path_msg.poses.append(pose_st)
    publisher.publish(path_msg)

def publish_smoothed_path(self, traj, publisher):
    # publish planned plan for visualization
    path_msg = Path()
    path_msg.header.frame_id = "map"
    for i in range(traj.shape[0]):
        pose_st = PoseStamped()
        pose_st.pose.position.x = traj[i, 0]
        pose_st.pose.position.y = traj[i, 1]
        pose_st.pose.orientation.w = 1
        pose_st.header.frame_id = "map"
        path_msg.poses.append(pose_st)
    publisher.publish(path_msg)

def publish_control(self):
    """
    Runs appropriate controller depending on the mode. Assumes all controllers
    are all properly set up / with the correct goals loaded
    """
    t = self.get_current_plan_time()

    if self.mode == Mode.PARK:
        V, om = self.pose_controller.compute_control(
            self.x, self.y, self.theta, t
        )
    elif self.mode == Mode.TRACK:
        V, om = self.traj_controller.compute_control(
            self.x, self.y, self.theta, t
```

AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

```
        )
    elif self.mode == Mode.ALIGN:
        V, om = self.heading_controller.compute_control(
            self.x, self.y, self.theta, t
        )
    else:
        V = 0.0
        om = 0.0

    cmd_vel = Twist()
    cmd_vel.linear.x = V
    cmd_vel.angular.z = om
    self.nav_vel_pub.publish(cmd_vel)

def get_current_plan_time(self):
    t = (rospy.get_rostime() - self.current_plan_start_time).to_sec()
    return max(0.0, t)  # clip negative time to 0

def replan(self):
    """
    loads goal into pose controller
    runs planner based on current pose
    if plan long enough to track:
        smooths resulting traj, loads it into traj_controller
        sets self.current_plan_start_time
        sets mode to ALIGN
    else:
        sets mode to PARK
    """
    # Make sure we have a map
    if not self.occupancy:
        rospy.loginfo(
            "Navigator: replanning canceled, waiting for occupancy map."
        )
        self.switch_mode(Mode.IDLE)
        return

    # Attempt to plan a path
    state_min = self.snap_to_grid((-self.plan_horizon, -self.plan_horizon))
    state_max = self.snap_to_grid((self.plan_horizon, self.plan_horizon))
    x_init = self.snap_to_grid((self.x, self.y))
    self.plan_start = x_init
```

```
x_goal = self.snap_to_grid((self.x_g, self.y_g))
problem = AStar(
    state_min,
    state_max,
    x_init,
    x_goal,
    self.occupancy,
    self.plan_resolution,
)

rospy.loginfo("Navigator: computing navigation plan")
success = problem.solve()
if not success:
    rospy.loginfo("Planning failed")
    return
rospy.loginfo("Planning Succeeded")

planned_path = problem.path

# Check whether path is too short
if len(planned_path) < 4:
    rospy.loginfo("Path too short to track")
    self.pose_controller.load_goal(self.x_g, self.y_g, self.theta_g)
    self.switch_mode(Mode.PARK)
    return

# Smooth and generate a trajectory
t_new, traj_new = compute_smoothed_traj(
    planned_path, self.v_des, self.spline_deg, self.spline_alpha, self.traj_dt
)

# If currently tracking a trajectory, check whether new trajectory will take more time to follow
if self.mode == Mode.TRACK:
    t_remaining_curr = (
        self.current_plan_duration - self.get_current_plan_time()
    )

    # Estimate duration of new trajectory
    th_init_new = traj_new[0, 2]
    th_err = wrapToPi(th_init_new - self.theta)
    t_init_align = abs(th_err / self.om_max)
```

AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

```
        t_remaining_new = t_init_align + t_new[-1]

        if t_remaining_new > t_remaining_curr:
            rospy.loginfo(
                "New plan rejected (longer duration than current plan)"
            )
            self.publish_smoothed_path(
                traj_new, self.nav_smoothed_path_rej_pub
            )
            return

    # Otherwise follow the new plan
    self.publish_planned_path(planned_path, self.nav_planned_path_pub)
    self.publish_smoothed_path(traj_new, self.nav_smoothed_path_pub)

    self.pose_controller.load_goal(self.x_g, self.y_g, self.theta_g)
    self.traj_controller.load_traj(t_new, traj_new)

    self.current_plan_start_time = rospy.get_rostime()
    self.current_plan_duration = t_new[-1]

    self.th_init = traj_new[0, 2]
    self.heading_controller.load_goal(self.th_init)

    if not self.aligned():
        rospy.loginfo("Not aligned with start direction")
        self.switch_mode(Mode.ALIGN)
        return

    rospy.loginfo("Ready to track")
    self.switch_mode(Mode.TRACK)

def run(self):
    rate = rospy.Rate(10)  # 10 Hz
    while not rospy.is_shutdown():
        # try to get state information to update self.x, self.y, self.theta
        try:
            (translation, rotation) = self.trans_listener.lookupTransform(
                "/map", "/base_footprint", rospy.Time(0)
            )
            self.x = translation[0]
            self.y = translation[1]
```

AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

```
        euler = tf.transformations.euler_from_quaternion(rotation)
        self.theta = euler[2]
    except (
        tf.LookupException,
        tf.ConnectivityException,
        tf.ExtrapolationException,
    ) as e:
        self.current_plan = []
        rospy.loginfo("Navigator: waiting for state info")
        self.switch_mode(Mode.IDLE)
        print(e)
        pass

    # STATE MACHINE LOGIC
    # some transitions handled by callbacks
    if self.mode == Mode.IDLE:
        pass
    elif self.mode == Mode.ALIGN:
        if self.aligned():
            self.current_plan_start_time = rospy.get_rostime()
            self.switch_mode(Mode.TRACK)
    elif self.mode == Mode.TRACK:
        if self.near_goal():
            self.switch_mode(Mode.PARK)
        elif not self.close_to_plan_start():
            rospy.loginfo("replanning because far from start")
            self.replan()
        elif (
            rospy.get_rostime() - self.current_plan_start_time
        ).to_sec() > self.current_plan_duration:
            rospy.loginfo("replanning because out of time")
            self.replan()  # we aren't near the goal but we thought we should have been, so
replan
    elif self.mode == Mode.PARK:
        if self.at_goal():
            # forget about goal:
            self.x_g = None
            self.y_g = None
            self.theta_g = None
            self.switch_mode(Mode.IDLE)

    self.publish_control()
```

AA274A Section 7
Writeup

Josie Oetjen, Karthik Pythireddi, Gerry Della Rocca

```
        rate.sleep()


if __name__ == "__main__":
    nav = Navigator()
    rospy.on_shutdown(nav.shutdown_callback)
    nav.run()
```