Muhamad Andi Darmawan
1103204182

Technical Report : Probability Mapping

Code :

```
import rclpy
import numpy as np
import threading

from rclpy.node import Node
from rclpy.qos import DurabilityPolicy, HistoryPolicy, QoSProfile
from nav_msgs.msg import OccupancyGrid
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import TransformStamped
from tf2_ros import StaticTransformBroadcaster, TransformListener, Buffer
from tf2_ros import LookupException, ConnectivityException
from tf2_ros import ExtrapolationException
from scipy.stats import norm
from scipy.interpolate import interp1d
from math import sin, cos, atan2, floor, degrees, isnan


ROBOT_DIAMETER = 7  # cm
WORLD_WIDTH = 3  # m
WORLD_HEIGHT = 3  # m
RESOLUTION = 0.01  # 1 cm

WORLD_ORIGIN_X = - WORLD_WIDTH / 2.0
WORLD_ORIGIN_Y = - WORLD_HEIGHT / 2.0
MAP_WIDTH = int(WORLD_WIDTH / RESOLUTION)
MAP_HEIGHT = int(WORLD_HEIGHT / RESOLUTION)

MIN_PROB = 0.01
MAX_PROB = 0.99
INITIAL_PROBABILITY = 0.50
L0 = 0.10

INFRARED_MAX = 0.07
TOF_MAX = 2.00

INFRARED_READING = [
   0.000,
   0.005,
   0.010,
```

```
    0.015,
    0.020,
    0.030,
    0.040,
    0.050,
    0.060,
    0.070
]

INFRARED_STD_DEV = [
    0.000,
    0.003,
    0.007,
    0.0406,
    0.01472,
    0.0241,
    0.0287,
    0.04225,
    0.03065,
    0.04897
]

TOF_READING = [
    0.00,
    0.05,
    0.10,
    0.20,
    0.50,
    1.00,
    1.70,
    2.00
]

TOF_STD_DEV = [
    0.126,
    0.032,
    0.019,
    0.009,
    0.007,
    0.010,
    0.013,
    0.000
]
```

```
class ProbabilityMapper(Node):
    def __init__(self, name):
        super().__init__(name)

        self._infra_red_interpolation = interp1d(INFRARED_READING,
                                             INFRARED_STD_DEV)
        self._tof_interpolation = interp1d(TOF_READING, TOF_STD_DEV)
        self._map_lock = threading.Lock()

        fill_map_param = self.declare_parameter('fill_map', True)

        # Init map related elements
        self._probability_map = [self._log_odd(INITIAL_PROBABILITY)] \
            * MAP_WIDTH * MAP_HEIGHT
        self.map_publisher = self.create_publisher(
            OccupancyGrid,
            '/prob_map',
            qos_profile=QoSProfile(
                depth=1,
                durability=DurabilityPolicy.TRANSIENT_LOCAL,
                history=HistoryPolicy.KEEP_LAST,
            )
        )
        self.tf_publisher = StaticTransformBroadcaster(self)
        tf = TransformStamped()
        tf.header.stamp = self.get_clock().now().to_msg()
        tf.header.frame_id = 'map'
        tf.child_frame_id = 'odom'
        tf.transform.translation.x = 0.0
        tf.transform.translation.y = 0.0
        tf.transform.translation.z = 0.0
        self.tf_publisher.sendTransform(tf)

        # Init laser related elements
        if fill_map_param.value:
            self.tf_buffer = Buffer()
            self.tf_listener = TransformListener(self.tf_buffer, self)
            self.scanner_subscriber = self.create_subscription(LaserScan,
                                                               '/scan',
                                                               self.update_map,
                                                               1)

        # Publish initial map
```

```
        self.publish_map()

        # Publish map every second
        self.create_timer(1, self.publish_map)

    def publish_map(self):
        self._map_lock.acquire()
        now = self.get_clock().now()
        prob_map = [-1] * MAP_WIDTH * MAP_HEIGHT
        idx = 0
        for cell in self._probability_map:
            cell_prob = round(self._prob(cell), 2)
            prob_map[idx] = int(cell_prob*100)
            idx += 1

        msg = OccupancyGrid()
        msg.header.stamp = now.to_msg()
        msg.header.frame_id = 'map'
        msg.info.resolution = RESOLUTION
        msg.info.width = MAP_WIDTH
        msg.info.height = MAP_HEIGHT
        msg.info.origin.position.x = WORLD_ORIGIN_X
        msg.info.origin.position.y = WORLD_ORIGIN_Y
        msg.data = prob_map
        self.map_publisher.publish(msg)
        self._map_lock.release()

    def update_map(self, msg):
        if self._map_lock.locked():
            return
        else:
            self._map_lock.acquire()

        # Determine transformation of laser and robot in respect to odometry
        laser_rotation = None
        laser_translation = None
        try:
            tf = self.tf_buffer.lookup_transform('odom',
                                    msg.header.frame_id,
                                    msg.header.stamp)
            q = tf.transform.rotation
            laser_rotation = atan2(2.0 * (q.w * q.z + q.x * q.y), 1.0 - 2.0 *
                            (q.y * q.y + q.z * q.z))
            laser_translation = tf.transform.translation
```

```
except (LookupException, ConnectivityException,
        ExtrapolationException):
    self._map_lock.release()
    return

# Determine position of robot and laser
# in map coordinate frame (in meter)
world_robot_x = laser_translation.x + WORLD_ORIGIN_X  # robot's center
world_robot_y = laser_translation.y + WORLD_ORIGIN_Y  # robot's center
world_laser_xs = []
world_laser_ys = []
world_laser_prob = []
laser_range_angle = msg.angle_min + laser_rotation
sensor_angle = msg.angle_min
for laser_range in msg.ranges:
    if laser_range < msg.range_max and laser_range > msg.range_min:
        world_laser_x = world_robot_x + laser_range * \
            cos(laser_range_angle)
        world_laser_y = world_robot_y + laser_range * \
            sin(laser_range_angle)
        tof = (round(degrees(sensor_angle)) == 0)
        if tof and laser_range > TOF_MAX:
            laser_range = TOF_MAX
        elif not tof and laser_range > INFRARED_MAX:
            laser_range = INFRARED_MAX
        _, log_odd = self._get_prob(laser_range, tof)
        world_laser_xs.append(world_laser_x)
        world_laser_ys.append(world_laser_y)
        world_laser_prob.append(log_odd)
    laser_range_angle += msg.angle_increment
    sensor_angle += msg.angle_increment

# Determine position on map (in cm)
robot_x = int(world_robot_x / RESOLUTION)
robot_y = int(world_robot_y / RESOLUTION)
laser_xs = []
laser_ys = []
laser_prob = world_laser_prob
for world_laser_x, world_laser_y in \
        zip(world_laser_xs, world_laser_ys):
    laser_xs.append(int(world_laser_x / RESOLUTION))
    laser_ys.append(int(world_laser_y / RESOLUTION))

# Fill the map based on known readings
```

```
        for laser_x, laser_y, prob in zip(laser_xs, laser_ys, laser_prob):
            index = laser_y * MAP_WIDTH + laser_x
            self.plot_bresenham_line(robot_x, laser_x, robot_y, laser_y, index)
            current_prob = self._probability_map[index]
            new_prob = current_prob + prob - self._log_odd(L0)
            self._probability_map[index] = new_prob

        # robot footprint
        start_x = robot_x - (floor(ROBOT_DIAMETER/2))
        start_y = robot_y - (floor(ROBOT_DIAMETER/2))
        for x in range(ROBOT_DIAMETER):
            for y in range(ROBOT_DIAMETER):
                index = (start_y + y) * MAP_WIDTH + (start_x + x)
                if (index >= 0 and index < len(self._probability_map)) or \
                   (index < 0 and index >= -len(self._probability_map)):
                    self._probability_map[index] = self._log_odd(MIN_PROB)
        self._map_lock.release()

    def plot_bresenham_line(self, x0, x1, y0, y1, idx):
        # Bresenham's line algorithm
        # (https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm)
        dx = abs(x1 - x0)
        sx = 1 if x0 < x1 else -1
        dy = -abs(y1 - y0)
        sy = 1 if y0 < y1 else -1
        err = dx + dy
        while True:
            index = y0 * MAP_WIDTH + x0
            if index != idx:
                current_prob = self._probability_map[index]
                new_prob = current_prob + self._log_odd(MIN_PROB) - \
                    self._log_odd(L0)
                self._probability_map[index] = new_prob
            if x0 == x1 and y0 == y1:
                break
            e2 = 2 * err
            if e2 >= dy:
                err += dy
                x0 += sx
            if e2 <= dx:
                err += dx
                y0 += sy

    @staticmethod
```

```
    def _prob(log_odd: float) -> float:
        result = np.exp(log_odd) / (1.0 + np.exp(log_odd))
        if isnan(result):
            result = 0.0
        return result

    @staticmethod
    def _log_odd(prob: float) -> float:
        return np.log(prob / (1.0 - prob))

    def _get_mean_and_std_dev(self, reading: float, tof: bool) -> tuple:
        assert reading >= 0.0, f"reading={reading}"

        if tof:
            lookup_table = self._tof_interpolation
        else:
            lookup_table = self._infra_red_interpolation

        return reading, lookup_table(reading)

    def _get_prob(self, reading: float, tof: bool) -> tuple:
        mean, std_dev = self._get_mean_and_std_dev(reading, tof)
        normal_dist = norm(loc=mean, scale=std_dev)
        probability = normal_dist.pdf(reading)

        if probability >= 1.0:
            probability = MAX_PROB
        elif probability == 0:
            probability = MIN_PROB

        log_odd = self._log_odd(probability)

        return probability, log_odd


def main(args=None):
    rclpy.init(args=args)
    epuck_mapper = ProbabilityMapper('probability_mapper')
    rclpy.spin(epuck_mapper)
    epuck_mapper.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
```

main()

Code ini adalah sebuah implementasi dari sebuah Node ROS (Robot Operating System) yang disebut "ProbabilityMapper". Node ini bertanggung jawab untuk memetakan probabilitas penghunian ruang (occupancy) berdasarkan data sensor laser dan sensor inframerah.

Pada awal kode, terdapat beberapa impor library dan modul yang diperlukan untuk menjalankan node ini, seperti rclpy untuk menghubungkan dengan ROS, numpy untuk operasi numerik, threading untuk penguncian (locking) saat mengakses data peta, dan beberapa pesan ROS seperti OccupancyGrid, LaserScan, dan TransformStamped.

Kemudian, terdapat beberapa konstanta dan variabel global yang didefinisikan, seperti ukuran peta dunia (WORLD_WIDTH dan WORLD_HEIGHT), resolusi peta (RESOLUTION), batas-batas probabilitas (MIN_PROB dan MAX_PROB), konfigurasi sensor inframerah dan sensor waktu terbang (ToF), serta beberapa variabel terkait peta dan pengukuran sensor.

Selanjutnya, terdapat definisi kelas ProbabilityMapper yang merupakan subclass dari Node dalam rclpy. Kelas ini berisi method dan logika yang digunakan untuk memetakan probabilitas penghunian ruang.

Beberapa method yang ada di dalam kelas ini antara lain:

__*init*__(): Method ini merupakan method konstruktor yang dipanggil saat objek ProbabilityMapper dibuat. Method ini melakukan inisialisasi variabel dan parameter yang dibutuhkan, seperti mempersiapkan penerbit (publisher) untuk mempublikasikan peta, menginisialisasi peta probabilitas awal, dan menyiapkan transformasi stasioner (static transform) antara peta dan odometer.

publish_map(): Method ini digunakan untuk mempublikasikan peta probabilitas saat ini. Peta probabilitas dikirim sebagai pesan OccupancyGrid melalui penerbit map_publisher.

update_map(): Method ini dipanggil saat ada pembaruan data sensor laser. Method ini mengupdate peta probabilitas berdasarkan data laser yang diterima. Data laser dikonversi ke koordinat peta dan nilai probabilitas yang sesuai, kemudian peta diisi dengan nilai probabilitas yang baru.

plot_bresenham_line(): Method ini digunakan untuk menggambar garis Bresenham antara dua titik pada peta. Method ini mengubah nilai probabilitas di sepanjang garis sesuai dengan nilai probabilitas yang ditentukan.

_prob(): Method ini menghitung probabilitas berdasarkan nilai log-odd yang diberikan.

_log_odd(): Method ini menghitung nilai log-odd berdasarkan probabilitas yang diberikan.

_get_mean_and_std_dev(): Method ini mengembalikan nilai rata-rata dan standar deviasi berdasarkan nilai bacaan sensor yang diberikan.

_get_prob(): Method ini mengembalikan probabilitas dan log-odd berdasarkan nilai bacaan sensor yang diberikan.

Terakhir, terdapat fungsi main() yang merupakan entry point dari program. Fungsi ini melakukan inisialisasi node ProbabilityMapper, menjalankan loop rclpy.spin() untuk menjaga