```
"""pf_controller controller."""

import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm
import math
import sys
import copy

from controller import Robot
from controller import Supervisor
from controller import Keyboard

MAX_SPEED = 12.3

################################## HELPER FUNCTIONS ###################################


# Normalizes the angle theta in range (-pi, pi)
def normalize_angle(theta):
    if (theta > np.pi):
        return theta - 2*np.pi
    if (theta < -np.pi):
        return theta + 2*np.pi
    return theta


norm_angle_arr = np.vectorize(normalize_angle)


def velFromKeyboard(keyboard):
    turn_base = 3.0
    linear_base = 6.0
    vel_left = 0.0
    vel_right = 0.0
    key = keyboard.getKey()
    while (key != -1):
        if (key==Keyboard.UP):
            vel_left += linear_base
            vel_right += linear_base
        if (key==Keyboard.DOWN):
            vel_left += -linear_base
            vel_right += -linear_base
        if (key==Keyboard.LEFT):
            vel_left += -turn_base
            vel_right += turn_base
        if (key==Keyboard.RIGHT):
            vel_left += turn_base
            vel_right += -turn_base
        key = keyboard.getKey()
    return vel_left, vel_right


# calculate the mean pose of a particle set.
#
# for x and y, the mean position is the mean of the particle coordinates
#
# for theta, we cannot simply average the angles because of the wraparound
# (jump from -pi to pi). Therefore, we generate unit vectors from the
```

```
# angles and calculate the angle of their average
def mean_pose(particles):
    # save x and y coordinates of particles
    xs = []
    ys = []

    # save unit vectors corresponding to particle orientations
    vxs_theta = []
    vys_theta = []

    for particle in particles:
        xs.append(particle[0])
        ys.append(particle[1])

        #make unit vector from particle orientation
        vxs_theta.append(np.cos(particle[2]))
        vys_theta.append(np.sin(particle[2]))

    #calculate average coordinates
    mean_x = np.mean(xs)
    mean_y = np.mean(ys)
    mean_theta = np.arctan2(np.mean(vys_theta), np.mean(vxs_theta))

    return [mean_x, mean_y, mean_theta]


# Visualizes the state of the particle filter.
#
# Displays the particle cloud, mean position and landmarks.
def plot_state(particles, obstacles, map_limits):
    xs = []
    ys = []


    for particle in particles:
        xs.append(particle[0])
        ys.append(particle[1])

    # mean pose as current estimate
    estimated_pose = mean_pose(particles)

    # plot filter state
    plt.clf()
    plt.axes().set_aspect('equal')
    plt.plot(xs, ys, 'r.')

    plt.quiver(estimated_pose[0], estimated_pose[1], np.cos(estimated_pose[2]),
np.sin(estimated_pose[2]), angles='xy',
               scale_units='xy')

    fig = plt.gcf()
    axes = fig.gca()
    for obs in obstacles:
        circle_plot = plt.Circle((obs[0], obs[1]), radius=obs[2], color='black',
zorder=1)
        axes.add_patch(circle_plot)
    plt.axis(map_limits)
    plt.pause(0.01)
```

```
def get_curr_pose(trans_field, rot_field):
    values = trans_field.getSFVec3f()
    rob_theta = np.sign(rot_field.getSFRotation()[2])*rot_field.getSFRotation()[3]
    rob_x = values[0]
    rob_y = values[1]
    return [rob_x, rob_y, rob_theta]


def get_pose_delta(last_pose, curr_pose):
    trans_delta = np.sqrt((last_pose[0]-curr_pose[0])**2 + (last_pose[1]-
curr_pose[1])**2)
    theta_delta = abs(normalize_angle(last_pose[2]-curr_pose[2]))
    return trans_delta, theta_delta


# Returns the odometry measurement between two poses
# according to the odometry-based motion model.
def get_odometry(last_pose, curr_pose):
    x = last_pose[0]
    y = last_pose[1]
    x_bar = curr_pose[0]
    y_bar = curr_pose[1]
    delta_trans = np.sqrt((x_bar - x) ** 2 + (y_bar - y) ** 2)
    delta_rot = normalize_angle(last_pose[2] - curr_pose[2])
    delta_rot1 = delta_rot / 2.0
    delta_rot2 = delta_rot / 2.0

    if (delta_trans > 0.01):
        delta_rot1 = normalize_angle(math.atan2(y_bar - y, x_bar - x) -
last_pose[2])
        delta_rot2 = normalize_angle(curr_pose[2] - last_pose[2] - delta_rot1)

    return [delta_rot1, delta_rot2, delta_trans]

def get_wheel_odometry(prev_readings, curr_readings):
    # follow the differential drive equations
    l = 0.34
    r = 0.195/2
    d_wheels = curr_readings - prev_readings
    R = (l/2)*(d_wheels[0] + d_wheels[1])/(d_wheels[1] - d_wheels[0])

    # since we are only interested in odometry, we can assume that the robot
    # starts at the origin with orientation theta = 0
    # with that we get a simpler expression of the ICC
    ICC = np.array([0.0, R])
    d_w = r*(d_wheels[1] - d_wheels[0])/l

    rot_mat = np.array([[np.cos(d_w), -np.sin(d_w)],
                        [np.sin(d_w), np.cos(d_w)]])
    new_position = rot_mat.dot(-ICC) + ICC
    last_pose = [0.0, 0.0, 0.0]
    curr_pose = [new_position[0], new_position[1], d_w]

    return get_odometry(last_pose, curr_pose)



##################################### SENSOR MODEL
```

```
##############################################


def beam_circle_intersection(A, B, C, r):
    d_0 = np.abs(C)
    if d_0 > r:
        return np.array([])
    x_0 = -A*C
    y_0 = -B*C
    if math.isclose(d_0, r):
        return np.array([[x_0, y_0]])
    d = np.sqrt(r*r - C*C)
    x_1 = x_0 + B*d
    y_1 = y_0 - A*d
    x_2 = x_0 - B*d
    y_2 = y_0 + A*d
    return np.array([[x_1, y_1], [x_2, y_2]])


"""
returns the distance to the closest intersection between a beam and an array of
circles
the beam starts at (x,y) and has the angle theta (rad) to the x-axis
circles is a numpy array with the structure [circle1, circle2,...]
where each element is a numpy array [x_c, y_c, r] describing a circle
with the center (x_c, y_c) and radius r
"""
def distance_to_closest_intersection(x, y, theta, circles):
    beam_dir_x = np.cos(theta)
    beam_dir_y = np.sin(theta)
    min_dist = float('inf')
    # iterate over all circles
    for circle in circles:
        # compute the line equation parameters for the beam
        # in a shifted coordinate system, with the circle center at origin
        x_shifted = x - circle[0]
        y_shifted = y - circle[1]
        # vector (A,B) is a normal vector orthogonal to the beam
        A = beam_dir_y
        B = -beam_dir_x
        C = -(A*x_shifted + B*y_shifted)
        intersections = beam_circle_intersection(A, B, C, circle[2])
        for isec in intersections:
            # check if intersection is in front of the robot
            dot_prod = (isec[0]-x_shifted)*beam_dir_x + (isec[1]-
y_shifted)*beam_dir_y
            if dot_prod > 0.0:
                dist = np.sqrt(np.square(isec[0]-x_shifted)+np.square(isec[1]-
y_shifted))
                if dist < min_dist:
                    min_dist = dist

    return min_dist


# Returns the distance to the closest intersection of a beam
# and the map borders. The beam starts at (x, y) and has
# the angle theta to the positive x-axis.
# The borders are given by map_limits = [x_min, x_max, y_min, y_max].
```

```
def distance_to_closest_border(x, y, theta, map_limits):
    ct = np.cos(theta)
    st = np.sin(theta)
    x_closest = float('inf')
    y_closest = float('inf')
    if (ct != 0.0):
        x_closest = max((map_limits[0]-x)/ct, (map_limits[1]-x)/ct)
    if (st != 0.0):
        y_closest = max((map_limits[2]-y)/st, (map_limits[3]-y)/st)
    return min(x_closest, y_closest)


# Returns the expected range measurements for all beams
# given the robot pose (x, y, theta_rob)
# and the map, described by circles and map_limits
def get_z_exp(x, y, theta_rob, n_beams, z_max, circles, map_limits):
    beam_directions = norm_angle_arr(np.linspace(-np.pi/2, np.pi/2, n_beams) +
theta_rob)
    z_exp = []
    for theta in beam_directions:
        dist = distance_to_closest_intersection(x, y, theta, circles)
        if (dist > z_max):
            dist = distance_to_closest_border(x, y, theta, map_limits)
        z_exp.append(dist)
    return z_exp


"""
z_scan and z_scan_exp are numpy arrays containing the measured and expected range
values (in cm)
b is the variance parameter of the measurement noise
z_max is the maximum range (in cm)
returns the probability of the scan according to the simplified beam-based model
"""
def beam_based_model(z_scan, z_scan_exp, b, z_max):
    prob_scan = 1.0
    alpha = 0.8
    p_random = 1.0/z_max
    for i in range(len(z_scan)):
        if z_scan[i] > z_max:
            continue
        prob_z = norm.pdf(z_scan[i], z_scan_exp[i], np.sqrt(b))
        prob_z = (1.0-alpha)*p_random + alpha*prob_z
        prob_scan *= prob_z
    return prob_scan


def eval_sensor_model(scan, particles, obstacles, map_limits):
    n_beams = len(scan)
    z_max = 10.0
    std_dev = 1.2
    var = std_dev**2
    weights = []
    for particle in particles:
        z_exp = get_z_exp(particle[0], particle[1], particle[2], n_beams, z_max,
obstacles, map_limits)
        weight = beam_based_model(scan, z_exp, var, z_max)
        weights.append(weight)
```

```
    weights = weights / sum(weights)

    return weights


################################### MOTION MODEL
##################################################


# Samples new particle positions, based on old positions, the odometry
# measurements and the motion noise
# (odometry based motion model)
def sample_motion_model(odometry, particles):
    delta_rot1 = odometry[0]
    delta_trans = odometry[2]
    delta_rot2 = odometry[1]

    # the motion noise parameters: [alpha1, alpha2, alpha3, alpha4]
    noise = [0.5, 0.5, 0.2, 0.2]

    # standard deviations of motion noise
    sigma_delta_rot1 = noise[0] * abs(delta_rot1) + noise[1] * delta_trans
    sigma_delta_trans = noise[2] * delta_trans + noise[3] * (abs(delta_rot1) +
abs(delta_rot2))
    sigma_delta_rot2 = noise[0] * abs(delta_rot2) + noise[1] * delta_trans

    # "move" each particle according to the odometry measurements plus sampled
noise
    # to generate new particle set
    new_particles = []

    for particle in particles:
        new_particle = [0.0, 0.0, 0.0]

        # sample noisy motions
        noisy_delta_rot1 = delta_rot1 + np.random.normal(0, sigma_delta_rot1)
        noisy_delta_trans = delta_trans + np.random.normal(0, sigma_delta_trans)
        noisy_delta_rot2 = delta_rot2 + np.random.normal(0, sigma_delta_rot2)

        # calculate new particle pose
        new_particle[0] = particle[0] + noisy_delta_trans * np.cos(particle[2] +
noisy_delta_rot1)
        new_particle[1] = particle[1] + noisy_delta_trans * np.sin(particle[2] +
noisy_delta_rot1)
        new_particle[2] = particle[2] + noisy_delta_rot1 + noisy_delta_rot2

        new_particles.append(new_particle)

    return new_particles


############################## PARTICLE SAMPLING
##################################################


# draw x,y and theta coordinate from uniform distribution
# inside map limits
def sample_random_particle(map_limits):
    x = np.random.uniform(map_limits[0], map_limits[1])
```

```
    y = np.random.uniform(map_limits[2], map_limits[3])
    theta = np.random.uniform(-np.pi, np.pi)

    return [x, y, theta]


# randomly initialize the particles inside the map limits
def initialize_particles(num_particles, map_limits):
    particles = []
    for i in range(num_particles):
        particles.append(sample_random_particle(map_limits))

    return particles



# Returns a new set of particles obtained by performing
# stochastic universal sampling, according to the particle weights.
# Further returns a list of weights for the newly sampled particles.
# The weight of a new particle is the same as the weight from which
# it was sampled.
def resample_particles(particles, weights):
    new_particles = []
    new_weights = []

    # distance between pointers
    step = 1.0 / len(particles)
    # random start of first pointer
    u = np.random.uniform(0, step)
    # where we are along the weights
    c = weights[0]
    # index of weight container and corresponding particle
    i = 0

    # loop over all particle weights
    for particle in particles:

        # go through the weights until you find the particle
        # to which the pointer points
        while u > c:
            i = i + 1
            c = c + weights[i]

        # add that particle
        new_particles.append(particles[i])
        new_weights.append(weights[i])

        # increase the threshold
        u = u + step

    return new_particles, new_weights


# Replaces half of the particles with the lowest weights
# with randomly sampled ones.
# Returns the new set of particles
def add_random_particles(particles, weights, map_limits):
    weight_indices = np.argsort(np.array(weights))
    n = len(particles)
```

```
    for i in range(n//2):
        sort_idx = weight_indices[i]
        particles[sort_idx] = sample_random_particle(map_limits)
    return particles


##############################################main#################################
def main():
    # create the Robot instance.
    robot = Supervisor()
    robot_node = robot.getFromDef("Pioneer3dx")

    # robot pose translation and rotation objects
    trans_field = robot_node.getField("translation")
    rot_field = robot_node.getField("rotation")

    # get the time step of the current world.
    timestep = int(robot.getBasicTimeStep())

    # init keyboard readings
    keyboard = Keyboard()
    keyboard.enable(10)

    # get wheel motor controllers
    leftMotor = robot.getDevice('left wheel')
    rightMotor = robot.getDevice('right wheel')
    leftMotor.setPosition(float('inf'))
    rightMotor.setPosition(float('inf'))

    # get wheel encoder sensors
    leftSensor = robot.getDevice('left wheel sensor')
    rightSensor = robot.getDevice('right wheel sensor')
    leftSensor.enable(60)
    rightSensor.enable(60)

    # initialize wheel velocities
    leftMotor.setVelocity(0.0)
    rightMotor.setVelocity(0.0)

    # get and enable lidar
    lidar = robot.getDevice('Sick LMS 291')
    lidar.enable(60)
    lidar.enablePointCloud()

    # create list of obstacles
    n_obs = 3
    obstacles = []
    for i in range(n_obs):
        obs_name = "Obs_"+str(i+1)
        obs_node = robot.getFromDef(obs_name)
        tr_field = obs_node.getField("translation")
        r_field = obs_node.getField("radius")
        x = tr_field.getSFVec3f()[0]
        y = tr_field.getSFVec3f()[1]
        r = r_field.getSFFloat()
        obstacles.append([x, y, r])

    # get map limits
    ground_node = robot.getFromDef("RectangleArena")
```

```
floor_size_field = ground_node.getField("floorSize")
fs_x = floor_size_field.getSFVec2f()[0]
fs_y = floor_size_field.getSFVec2f()[1]
map_limits = [-fs_x/2.0, fs_x/2.0, -fs_y/2.0, fs_y/2.0]

# init particles and weights
n_particles = 100
particles = initialize_particles(n_particles, map_limits)
weights = [1.0/n_particles]*n_particles

# last pose used for odometry calculations
last_pose = get_curr_pose(trans_field, rot_field)
last_wheel_meas = np.array([leftSensor.getValue(), rightSensor.getValue()])
# translation threshold for odometry calculation
trans_thr = 0.1


while robot.step(timestep) != -1:
    # key controls
    vel_left, vel_right = velFromKeyboard(keyboard)
    leftMotor.setVelocity(vel_left)
    rightMotor.setVelocity(vel_right)

    # read robot pose and compute difference to last used pose
    curr_pose = get_curr_pose(trans_field, rot_field)
    trans_delta, theta_delta = get_pose_delta(last_pose, curr_pose)

    # skip until translation change is big enough
    if (trans_delta < trans_thr):
        continue

    # get current lidar measurements
    scan = lidar.getRangeImage()
    # we use a reversed scan order in the sensor model
    scan.reverse()

    # compute odometry from pose difference
    #odometry_gt = get_odometry(last_pose, curr_pose)
    curr_wheel_meas = np.array([leftSensor.getValue(), rightSensor.getValue()])
    odometry = get_wheel_odometry(last_wheel_meas, curr_wheel_meas)

    last_wheel_meas = curr_wheel_meas
    last_pose = curr_pose

    # insert random particles
    particles = add_random_particles(particles, weights, map_limits)

    # predict particles by sampling from motion model with odometry info
    particles = sample_motion_model(odometry, particles)

    #calculate importance weights according to sensor model
    weights = eval_sensor_model(scan, particles, obstacles, map_limits)

    #resample new particle set according to their importance weights
    particles, weights = resample_particles(particles, weights)

    # plot current particle state
    plot_state(particles, obstacles, map_limits)
```

```
    plt.show('hold')


if __name__ == "__main__":
    main()
```