```python
"""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

from misc_tools import *

MAX_SPEED = 12.3


################## SIMULATOR HELPERS ##########################################

# 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

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


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_sensor_reading(landmarks, pose):
    px = pose[0]
    py = pose[1]

    lm_ids = []
    x = []
    y = []

    for i in range(len(landmarks)):
        lx_clean = landmarks[i+1][0]
        ly_clean = landmarks[i+1][1]
        noise = np.random.normal(loc=0.0, scale=0.5, size=2)
        lx = lx_clean + noise[0]
        ly = ly_clean + noise[1]
        id = i+1

        # calculate expected range measurement
        meas_range = np.sqrt( (lx - px )**2 + (ly - py )**2 )


        if meas_range > 5.0:
            continue

        lm_ids.append(id)
        x.append(lx-px)
        y.append(ly-py)

    return {'id':lm_ids,'x':x,'y':y}

def initialize_particles(num_particles, num_landmarks, init_pose):
    #initialize particle at pose [0,0,0] with an empty map

    particles = []
```

```
    for i in range(num_particles):
        particle = dict()

        #initialize pose: at the beginning, robot is certain it is at [0,0,0]
        particle['x'] = init_pose[0]
        particle['y'] = init_pose[1]
        particle['theta'] = init_pose[2]

        #initial weight
        particle['weight'] = 1.0 / num_particles

        #particle history aka all visited poses
        particle['history'] = []

        #initialize landmarks of the particle
        landmarks = dict()

        for i in range(num_landmarks):
            landmark = dict()

            #initialize the landmark mean and covariance
            landmark['mu'] = [0,0]
            landmark['sigma'] = np.zeros([2,2])
            landmark['observed'] = False

            landmarks[i+1] = landmark

        #add landmarks to particle
        particle['landmarks'] = landmarks

        #add particle to set
        particles.append(particle)

    return particles


################### MEASUREMENT MODEL #########################################


def eval_sensor_model(measurements, particles):
    # Correct landmark poses with a measurement and
    # calculate particle weight

    # sensor noise
    R_t = np.array([[1.0, 0],
                    [0, 1.0]])

    # measured landmark ids and ranges
    ids = measurements['id']
    x_dists = measurements['x']
    y_dists = measurements['y']

    # update landmarks and calculate weight for each particle
    for particle in particles:

        landmarks = particle['landmarks']
        particle['weight'] = 1.0
```

```
        px = particle['x']
        py = particle['y']

        # loop over observed landmarks
        for i in range(len(ids)):

            # current landmark
            lm_id = ids[i]
            landmark = landmarks[lm_id]

            # measured range and bearing to current landmark
            x_dist = x_dists[i] + px
            y_dist = y_dists[i] + py

            if not landmark['observed']:
                # landmark is observed for the first time

                # initialize landmark position based on the measurement and
particle pose
                lx = x_dist
                ly = y_dist
                landmark['mu'] = [lx, ly]
                landmark['sigma'] = R_t
                landmark['observed'] = True

            else:
                # landmark was observed before

                C = np.identity(2)

                # Calculate measurement covariance and Kalman gain
                S = landmark['sigma']
                Q = C.dot(S).dot(C.T) + R_t
                K = S.dot(C.T).dot(np.linalg.inv(Q))

                # Compute the difference between the observed and the expected
measurement
                lm_arr = np.array(landmark['mu'])
                exp_meas = C.dot(lm_arr)
                delta = np.array([x_dist - exp_meas[0], y_dist-exp_meas[1]])

                # update estimated landmark position and covariance
                landmark['mu'] = landmark['mu'] + K.dot(delta)
                landmark['sigma'] = (np.identity(2) - K.dot(C)).dot(S)

                # compute the likelihood of this observation
                fact = 1 / np.sqrt(math.pow( 2 *math.pi ,2) * np.linalg.det(Q))
                expo = -0.5 * np.dot(delta.T, np.linalg.inv(Q)).dot(delta)
                weight = fact * np.exp(expo)

                # calculate overall weight, account for observing
                # multiple landmarks at one time step
                particle['weight'] = particle['weight'] * weight

    # normalize weights
    normalizer = sum([p['weight'] for p in particles])

    for particle in particles:
        particle['weight'] = particle['weight'] / normalizer
```

```
    return

###################### MOTION MODEL ###########################################
def sample_normal(mu, sigma):
  return np.random.normal(mu,sigma)


def sample_odometry_motion_model(x, u, a):
    """ Sample odometry motion model.

    Arguments:
    x -- pose of the robot before moving [x, y, theta]
    u -- odometry reading obtained from the robot [rot1, rot2, trans]
    a -- noise parameters of the motion model [a1, a2, a3, a4]

    """

    delta_hat_r1 = u[0] + sample_normal(0, a[0] * abs(u[0]) + a[1] * u[2])
    delta_hat_t = u[2] + sample_normal(0, a[2] * u[2] + a[3] * (
                abs(u[0]) + abs(u[1])))
    delta_hat_r2 = u[1] + sample_normal(0, a[0] * abs(u[1]) + a[1] * u[2])

    x_prime = x[0] + delta_hat_t * math.cos(x[2] + delta_hat_r1)
    y_prime = x[1] + delta_hat_t * math.sin(x[2] + delta_hat_r1)
    theta_prime = x[2] + delta_hat_r1 + delta_hat_r2

    return np.array([x_prime, y_prime, theta_prime])

def sample_motion_model(odometry, particles):
    # Samples new particle positions, based on old positions, the odometry
    # measurements and the motion noise
    # the motion noise parameters: [alpha1, alpha2, alpha3, alpha4]
    noise = [0.1, 0.1, 0.05, 0.05]
    for particle in particles:
        """
        signature of: sample_odometry_motion_model(x, u, a)
        x -- pose of the robot before moving [x, y, theta]
        u -- odometry reading obtained from the robot [rot1, rot2, trans]
        a -- noise parameters of the motion model [a1, a2, a3, a4]
        """
        particle['x'], particle['y'], particle['theta'] =
sample_odometry_motion_model(
            [particle['x'], particle['y'], particle['theta']],
            [odometry['r1'], odometry['r2'], odometry['t']],
            noise
        )

        # remember last position to draw path of particle
        particle['history'].append([particle['x'], particle['y']])

    return


##################### RESAMPLING ##############################################

def resample_particles(particles):
    # Returns a new set of particles obtained by performing
    # stochastic universal sampling, according to the particle
```

```
    # 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 = particles[0]['weight']

    # index of weight container and corresponding particle
    i = 0

    new_particles = []

    # 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 + particles[i]['weight']

        # add that particle
        new_particle = copy.deepcopy(particles[i])
        new_particle['weight'] = 1.0 / len(particles)
        new_particles.append(new_particle)

        # increase the threshold
        u = u + step

    return new_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'))
```

```python
# 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)

# create list of landmarks
n_obs = 9
landmarks = {}
for i in range(n_obs):
    obs_name = "Obs_"+str(i+1)
    obs_node = robot.getFromDef(obs_name)
    tr_field = obs_node.getField("translation")
    x = tr_field.getSFVec3f()[0]
    y = tr_field.getSFVec3f()[1]
    landmarks[i+1] = [x, y]

# 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
num_particles = 100
num_landmarks = len(landmarks)

# last pose used for odometry calculations
last_pose = get_curr_pose(trans_field, rot_field)

# particle initialization
particles = initialize_particles(num_particles, num_landmarks, last_pose)

# translation threshold for odometry calculation
trans_thr = 0.1

# initialize ground truth poses
gt_poses = [last_pose]
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

    # compute odometry
    odom_raw = get_odometry(last_pose, curr_pose)
```

```
        last_pose = curr_pose
        gt_poses.append(curr_pose)
        odom_dict = dict()
        odom_dict['r1'] = odom_raw[0]
        odom_dict['r2'] = odom_raw[1]
        odom_dict['t'] = odom_raw[2]

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

        # generate sensor measurements
        measurements = get_sensor_reading(landmarks, curr_pose)

        #evaluate sensor model to update landmarks and calculate particle weights
        eval_sensor_model(measurements, particles)

        #plot filter state
        plot_state(particles, landmarks, map_limits, gt_poses)

        #calculate new set of equally weighted particles
        particles = resample_particles(particles)


    plt.show('hold')


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