COMP462
Project 3: State Estimation

In this project, you are required to implement the Particle Filtering algorithm for estimating the 2D pose of the robot base of a 7-DoF manipulator by letting the robot touch the surrounding obstacles in the environment. A code framework, hosted on github https://github.com/robotpilab/COMP462/tree/main/project3, is provided to facilitate the implementation.

> **Report:** **A report is required for this project. The requirements of what to be included in the report are highlighted in the description.**

The simulation of the robot is provided, which is based on `pybullet`. As illustrated in Figure 1, a stick with one spherical end is fixed to the robot hand. While observing the environment, the robot needs to make contact with the surrounding obstacles by the spherical end of the stick.

Figure 1: The simulation of the 7-DoF robot manipulator by `pybullet`. A stick with a spherical end is fixed to the hand of the robot. The robot will make contact with the surrounding cylinder obstacles (green) through the spherical end of the stick.

There are three parameters to estimate: (x, y, θ), where (x, y) is the position of the robot base on the ground and θ is the orientation of the robot base about its z-axis (perpendicular to the ground). Each observation is represented by the robot configuration (i.e., joint angles) at which the spherical end of the stick is contacting with any obstacle.

Before starting the project, please first run "`python join_urdf.py`" in the terminal to prepare the necessary URDF file for the simulation.

Task 1: Particle Weight Calculation

In Particle Filtering, the state of a particle is represented by the parameters to estimate, and the weight of a particle is its probability of being resampled. In practice, the weight of particles will be updated by the likelihood of the current observation. In this part, you will be given 10 particles

and 1 observation, and you need to calculate the weights for each particle and give the most likely
one. The major steps for this task are detailed below:

1. First, implement the following function in `alg.py` to obtain the distance between the stick's
   spherical end centered at (x, y) to its closest cylinder obstacle:

   ```
   def dist_to_closest_obs(x, y)
       """
       Find the distance between the stick's sphere end centered at (x, y)
       to its closest cylinder obstacle.
       args:       x: The x-coordinate of the spherical end.
                   y: The y-coordinate of the spherical end.
       returns: dist: The distance to the closest obstacle.
       """
   ```

   It is worth noting that the distance between the sphere and the cylinder is the distance
   between their (side) surfaces, not centers. Such distance can be calculated by the distance
   between their centers subtracting their radiuses. When the sphere contacts the cylinder, the
   distance should be zero; and when the sphere penetrates the cylinder, the distance should be
   negative; otherwise, a positive distance is expected. The closest cylinder obstacle is the one
   whose distance from the sphere is closest to zero.

   There are 5 cylinder obstacles in the environment. You can access the 2D positions and
   radiuses of them in the variables `OBS_CENTER` and `OBS_RADIUS` respectively. And the radius
   of the stick's spherical end is in the variable `SPH_RADIUS` whose value is 0.02.

2. Next, you need to implement the following function in `alg.py` to evaluate the weights for the
   particles according to the given observation (i.e., the robot's joint angles when the spherical
   end of the stick is contacting one of the cylinder obstacles):

   ```
   def cal_weights(particles, obv, sigma=0.05)
       """
       Calculate the weights for particles based on the given observation.
       args:  particles: The particles represented by their states
                         Type: numpy.ndarray of shape (# of particles, 3)
                    obv: The given observation (the robot's joint angles).
                         Type: numpy.ndarray of shape (7,)
                  sigma: The standard deviation of the Gaussian distribution
                         for calculating likelihood (default: 0.05).
       returns: weights: The weights of all particles.
                         Type: numpy.ndarray of shape (# of particles,)
       """
   ```

   Suppose the number of particles is N, then the input variable `particles` in the above function
   should be $(N, 3)$, of which each row represents the state of one particle. If we denote the
   state of the i-th particle (i.e., the i-th row of the variable `particles`) by (x_i, y_i, θ_i), then the
   weight of this particle should be calculated by a Gaussian distribution:

   $$w_i = \frac{\varphi(d_i)}{\sum_{j=1}^{N} \varphi(d_j)}$$

   where $\varphi(\cdot)$ is the probability density function (pdf) of a Gaussian distribution with its stan-
   dard deviation being `sigma` (whose value is 0.05 by default); The denominator of the above
   equation is used to normalize the weights of the particles such that their sum is 1 (i.e.,
   $\sum_{i=1}^{N} w_i = 1$). This is because the weights will be used as the probabilities for resampling
   particles in the Particle Filtering algorithm.

   Assuming the robot base is at (x_i, y_i, θ_i) and the robot's joint angles are given by the observa-
   tion (i.e., the input variable `obv`), d_i in the above equation is the distance between the stick's

spherical end and its closest cylinder obstacle (calculated by your implemented function in Step 1).

To obtain the 2D position (i.e., x and y coordinates) of the stick's spherical end with respect to the robot base frame when the robot's joint angles are `obv`, you need forward kinematics of the robot. You can use the provided forward kinematics solver to do this calculation by `FK_Solver.forward_kinematics_2d(obv)`.

3. Finally, please complete the following function in `alg.py` to give the index of the most likely particle (i.e., the particle with the highest weight):

```
def most_likely_particle(particles, weights):
    """
    Find the most likely particle.
    args:  particles: The particles represented by their states
                      Type: numpy.ndarray of shape (# of particles, 3)
                 obv: The given observation (the robot's joint angles).
                      Type: numpy.ndarray of shape (7,)
           returns:   idx: The index of the most likely particle
                      Type: int
    """
```

After completing this part, you can run the following command in your terminal to test your program: `python main.py --task 1`, and the results of running your implementation will be printed in the terminal.

> **Report:** With `sigma = 0.05` (the standard deviation of the Gaussian), please report the weights of all particles and the index of the most likely particle.

Task 2: Estimation with Given Observations

For this part, you will be given 100 observations, that is, 100 different robot configurations when contacting the obstacles. You need to implement the Particle Filtering algorithm by iteratively updating the particles and their weights with each observation. Please finish the `TODO` in the following function in `alg.py`:

```
def particle_filter(panda_sim, obvs, num_particles, sigma=0.05)
    """
    The Particle Filtering algorithm.
    args:      panda_sim: The instance of the simulation.
                          Type: sim.PandaSim (provided)
                    obvs: The given observations (the robot's joint angles)
                          Type: numpy.ndarray of shape (# of observations, 7)
           num_particles: The number of particles.
                          Type: int
                   sigma: The standard deviation of the Gaussian distribution
                          for calculating likelihood (default: 0.05).
           returns:       est: The estimate of the pose of the robot base.
                          Type: numpy.ndarray of shape (3,)
    """
```

The particles are initialized by uniform random sampling in the space of the parameters - (x, y, θ), and the weight of each particle is equally initialized to be $\frac{1}{N}$ where N is the number of the particles. Then, you need to complete the rest of the Particle Filtering algorithm.

At each iteration of the algorithm, you need to use one of the given observations (i.e., one row of `obvs`) to update the particles and their weights, and resample the particles according to their weights. As the particles are updated and resampled iteratively, they will finally converge to

cluster around the groundtruth of (x, y, θ) of the robot base. And the estimate of (x, y, θ), which is calculated by the average of all particles, should be also close to the groundtruth.

Note: It is crucial to slightly perturb each particle at the beginning of each iteration of Particle Filtering. To do this, for each particle $p_i = (x_i, y_i, \theta_i)$, you can add some Gaussian randomness by:

$$(x_i, y_i, \theta_i) + \epsilon_i$$

In the above expression, ϵ_i is a multivariate Gaussian random variable, i.e., $\epsilon \sim \mathcal{N}(0, \sigma \mathbb{I}_{3 \times 3})$, where $\sigma = 0.01$ is the scale of the Gaussian distribution, and $\mathbb{I}_{3 \times 3}$ is the 3-by-3 identity matrix. This particle perturbation mechanism is very helpful for the robustness of the Particle Filtering algorithm in the robot calibration application. To see that, suppose no perturbation is applied and no randomness is added between iterations of the particle filtering algorithm. Then, when resampling particles, the same particle with a high weight can be sampled multiple times and the particles with low weights may not be sampled. As such, after resampling, there could be redundant particles and some other particles (especially those with low weights) could vanish. As the algorithm iterates, the number of different particles can gradually decrease to lose the distribution information of the parameters to estimate. However, by adding randomness between iterations, the particles can locally spread to effectively represent the distribution of the parameters, which is important to the robustness of the algorithm.
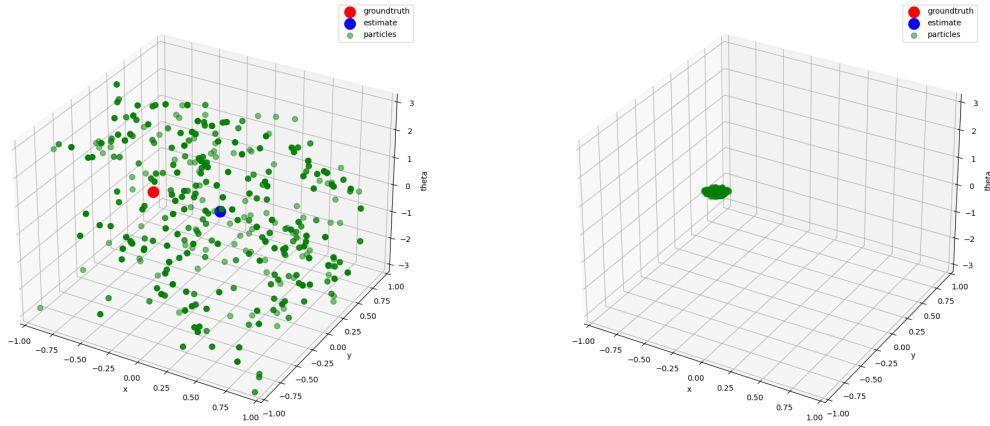

Figure 2: Visualization of the Particle Filtering algorithm in the parameter space, i.e., (x, y, θ). *Left:* At the beginning of Particle Filtering, the particles (green) are uniformly initialized, and the estimate (blue) of the parameters is not close to the groundtruth (red). *Right:* As the algorithm iterates, the particles will converge to the groundtruth.

After completing this part, you can run the following command in your terminal to test your program, which visualizes the robot configuration of each observation in the simulator and the particles in the parameter space shown in Figure 2: `python main.py --task 2`.

Moreover, to set the number of particles and the standard deviation of the Gaussian (i.e., `sigma`) for updating weights, you can just add `--num_particles the_number_of_particles` and `--sigma the_value_of_sigma` to the end of your command.

> **Report:** With 1000 and 2500 **particles, and with** `sigma` $= 0.01, 0.05, 0.2$ **respectively, run your implemented Particle Filtering algorithm. For each combination of** `sigma` **and the number of particles, you can run multiple trials. In your report, please discuss: how the number of particles and the value of** `sigma` **affect the performance of the Particle Filtering algorithm (e.g., the convergence speed, the robustness, etc)? And please report your estimate of the parameters, i.e., (x, y, θ) of the robot base.**

(Optional) Task 3: Estimation while Generating Observations

Instead of using the pre-collected observations and running Particle Filtering offline. In this part, you are asked to implement a method to collect observations online. For this task, it is recommended to complete the following steps:

1. First, please implement the following function in `alg.py` to control the robot manipulator in the simulation to collect an observation:

    ```
    def get_one_obv(panda_sim)
        """
        Control the robot in simulation to obtain an observation.
        args: panda_sim: The instance of the simulation.
                          Type: sim.PandaSim (provided)
        returns:    obv: One observation found by this function.
                          Type: numpy.ndarray of shape (7,)
        """
    ```

    For this step, you are recommended to briefly read the codes of the provided class `PandaSim` in `sim.py` to be familiar with the simulation environment, which is similar to the one in Homework 1. Particularly, there are three functions in this class worth noting:

    The function `PandaSim.execute()` is used to command the robot by an input Cartesian velocity. The robot will move its end-effector with the velocity you give by one simulation step (i.e., 0.002 seconds in simulation).

    The function `PandaSim.is_collision()` is used to detect collisions between the robot and the obstacles. This can be useful for safely commanding the robot to avoid unnatural behavior of the robot (e.g., penetration).

    The function `PandaSim.is_touch()` is used to detect if there is a contact between the stick's spherical end and any cylinder obstacle. You can use this function to determine when to obtain an observation.

2. Next, please complete the following function of online Particle Filtering in `alg.py`. The implementation of this function should be very similar to the one you completed in Task 2. The only difference is that here you need to integrate your implemented `get_one_obv()` function in Step 1 to use observations collected online.

    ```
    def particle_filter_online(panda_sim, num_particles, sigma=0.05)
        """
        The online Particle Filtering algorithm.
        args:      panda_sim: The instance of the simulation.
                              Type: sim.PandaSim (provided)
               num_particles: The number of particles.
                              Type: int
                       sigma: The standard deviation of the Gaussian distribution
                              for calculating likelihood (default: 0.05).
        returns:         est: The estimate of the pose of the robot base.
                              Type: numpy.ndarray of shape (3,)
        """
    ```

After completing this part, you can run the following command in your terminal to test your program, which will render the simulation to and visualize the particles in real time: `python main.py --task 3`.

Report: In the report, please describe your online Particle Filtering algorithm (including how you control the robot and collect observations online) with both text and pseudo codes. Please visualize the particles in your report to show the convergence of the particles. You can choose to use any hyperparameters (the number of particles, the number of iterations, the value of `sigma`) that you believe work best. Please report the values of these hyperparameters you use.