ROB 502 FA24: Homework 7

See Canvas for due dates.

Rules:

1. All HW must be done individually, but you are encouraged to post questions on Piazza and discuss during OHs (recall: do not email assignment questions to teaching staff!)
2. Late work adheres to the late grading policy (see syllabus, or use late-day tokens)
3. Submit your code on Autograder.io
4. Remember that copying-and-pasting code from other sources is not allowed

Code Download:
The code is available on the ROB 502 FA24 Git repo in the appropriate HW folder.

We prefer that you use the '`fetch`'/'`pull`' Git methods for the HWs, but we know that Git can be hard/confusing; if necessary, you can always just **download** the code from the repo link. However, learning to use Git will be to your benefit for multiple reasons:

1. You can get some easy extra credit if you properly pull and version-control (and comment) your work for the HW assignments!
2. ROB 550 will require you to use Git in a much more complex capacity, and without much learning time (large code databases, lots of teamwork with multiple branches, etc.).
3. If you ever need to setup a new VM or use your code on another device (new computer, VM crashes, etc.) you can easily restore your code progress with Git.
4. Many robotics-related industries and research labs rely on Git for robust version-controlling of their work… if mastered, you can put this on your resumé!

Open the directory `hw7` in VSCode. You can do this by `cd`-ing to the `hw7` directory and running `code .` . Do not run VSCode from subdirectories of `hw7` .

Instructions:

See previous homework assignments for notes about using Autograder.io, but you should be an expert now!

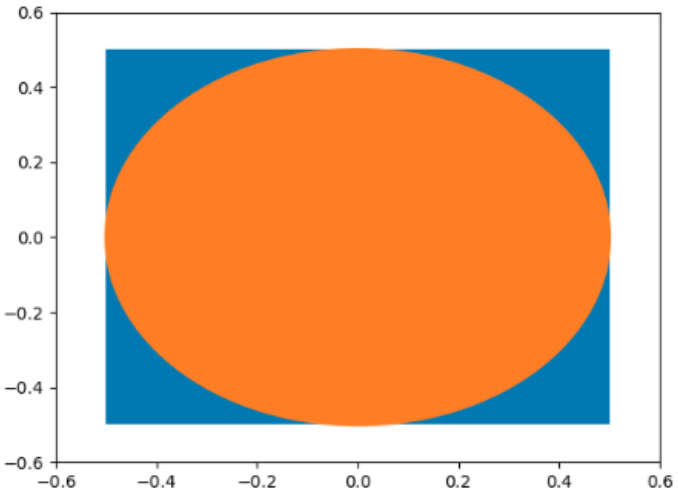
Each problem will give you a file with some template code, and you need to fill in the rest. Make sure to only put your code in the areas that start with `// --- Your code here` and end with `// ---`. **Do not edit code outside these blocks!**

Sample input and output are given as `input.txt` and `output.txt` respectively. Getting your program to successfully compile and correctly reproduce the sample output will get you some points, while the remaining points will be from hidden inputs that test some edge cases. Think about valid inputs that could break your logic. **Note that the templates given will read from `input.txt` and output to `output.txt`, so it might be a good idea to make a copy of the sample output to avoid overriding it.**

1. **Warmup with NumPy** [12 points]:

   You have two tasks in this problem:

   a. **Estimating PI with samples**
   Your task is to complete the function `estimate_pi` in `hw7/numpy_warmup/estimate_pi.py`. This function will estimate the value of π using random samples, using the `numpy.random` module. To estimate π, we can use the fact that a square of side length 1 has area of 1. In contrast, a circle with a diameter of 1 has an area of π/4.



   We use this fact to determine $\frac{A_{circle}}{A_{square}} = \frac{\pi}{4}$. Thus a method of estimating π is as follows:

   1. Sample N_{total} points uniformly from the unit square
   2. Check if those points are inside the unit circle (let N_{circle} be the number of sampled points in the circle)
   3. Your estimate of π is then $\pi = 4\frac{N_{circle}}{N_{total}}$

   Your function should take as input the value N_{total}, and output an estimate of π. You should notice that the estimate improves when you use more samples.

   You can test your code and by running `estimate_pi.py`, which will print the estimate of π to the terminal.

   b. **Vectorizing nested `for` loops**
   In Homework 5 you solved a QP which contained quadratic cost term $x^T Q x$. This quadratic cost term is extremely common in robotics. In planning & control, for example, it is common to have a quadratic cost term on the distance of the robot to a goal. In this context it is common to use a total cost which is the sum of quadratic costs:

$$Cost = \sum_{k=1}^{K} x_k^T \, Q \, x_k$$

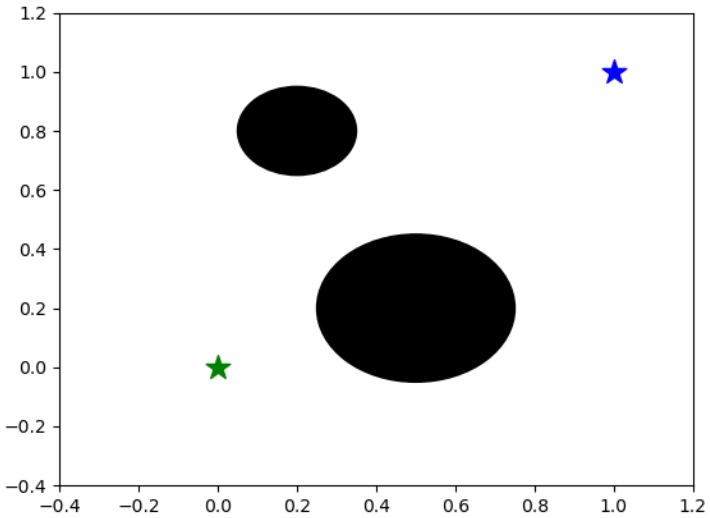
Open `hw7/numpy_warmup/quadratic_cost.py` . You will see that there is a function `quadratic_cost` already there, as follows:

```
def quadratic_cost(Q, x):
    K, d = x.shape
    cost_sum = 0
    for k in range(K):
        for i in range(d):
            for j in range(d):
                cost_sum += x[k][i] * Q[i][j] * x[k][j]
    return cost_sum
```

This function computes the total quadratic cost, but unfortunately has 3 nested for-loops. Your task is to use NumPy to "vectorize" this function. Vectorizing in NumPy is the process of replacing slow operations with fast array operations in NumPy. In this example you need to remove all `for` loops, replacing them with NumPy functions, to complete the function `quadratic_cost_vectorized`. You can run `quadratic_cost.py` which will compute costs and compare the times for the vectorized and non-vectorized versions using `time_function_call` . You are given two examples `example1.npz` and `example2.npz`. You should see a speed up of at least 3x and 50x for these examples, respectively, if your code is appropriately vectorized.

2. **Trajectory optimization with Genetic Algorithms** [22 points]:

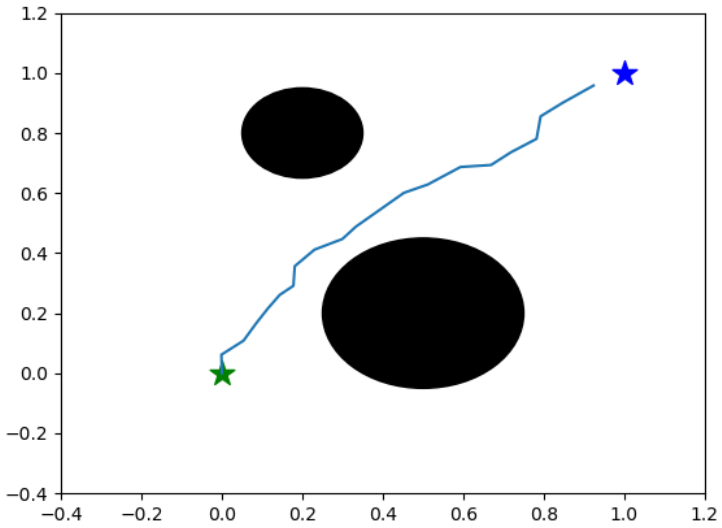
In this problem we will perform a trajectory optimization using a genetic algorithm. Our goal is to generate a path from the start (green star) to the goal (blue star) that avoids the spherical obstacles shown in black. The generated paths will be of length `T` and will be parameterized as a sequence of steps $(\Delta\mathbf{x}_1, \Delta\mathbf{x}_2, \ldots \Delta\mathbf{x}_T)$ where $\Delta\mathbf{x}_t = \mathbf{x}_t - \mathbf{x}_{t-1}$ and each $\mathbf{x}_t = (x_t, y_t)$, i.e., the 2D position. To find the position at any time `t` we compute $\mathbf{x}_t = \mathbf{x}_{start} + \sum_{k=1}^{t} \Delta\mathbf{x}_k$, and the final position as $\mathbf{x}_T = \mathbf{x}_{start} + \sum_{k=1}^{T} \Delta\mathbf{x}_k$.



The files given are `hw7/ga_trajectory_optimization/ga_trajopt.py` and `hw7/ga_trajectory_optimization/ga_trajopt_soln.py` .
You should only modify `hw7/ga_trajectory_optimization/ga_trajopt_soln.py`!

`hw7/ga_trajectory_optimization/ga_trajopt.py` contains the class `GATrajectoryOptimizer` with the following member functions:

- `mutate` which will use Gaussian noise to randomly mutate a solution from the population
- `tournament_selection` which selects `N` new parents from the population. More details about the tournament selection algorithm can be found here: https://en.wikipedia.org/wiki/Tournament_selection
- `generate_children` which, given a set of two parents, generates two children via a combination of mutation and crossover
- `solve` This is the main function that performs the GA optimization
- `plot_trajectory` this takes a sequence $(\Delta\mathbf{x}_1, \Delta\mathbf{x}_2, \ldots \Delta\mathbf{x}_T)$ and generates a plot. An example of a successful trajectory plotted is shown below:



`hw7/ga_trajectory_optimization/ga_trajopt_soln.py` contains the class `GATrajectoryOptimizerSolution` which contains member functions which **you should modify.** They are:

- `fitness` this computes the fitness of a sequence $(\Delta\mathbf{x}_1, \Delta\mathbf{x}_2, \ldots \Delta\mathbf{x}_T)$
- `crossover` this takes two parents and uses crossover to generate two new children
- `select_children` this takes a parent and a child and decides which to keep in the population

In addition under `if __name__ == '__main__'` in `hw7/ga_trajectory_optimization/ga_trajopt_soln.py` there is code which instantiates `GATrajectoryOptimizerSolution`, runs the solver and plots the result. To run the entire algorithm, you can run `python3 ga_trajopt_soln.py`.

To test your code you can run `ga_trajopt_tests.py`

Complete this problem in the following steps:

a. Read through the code we gave you to try to understand which functions have already been written, and what member variables you will need to read/write. This will make the rest of the problem much quicker and prevent you from accidentally re-implementing things we've already written. The ability to read through someone else's code and get a general understanding of what it does/should do is an essential skill! If you get confused, try making a diagram on paper that has every class (along with member functions) and the inheritance relationship between classes, then go through the code and see which file contains which function.

b. Complete `fitness` in `GATrajectoryOptimizerSolution`. Your code should use the following fitness function:

$$\text{Fitness} = -\left[\left\|\mathbf{x}_T - \mathbf{x}_{goal}\right\|_2^2 + \sum_{t=1}^{T}\|\Delta\mathbf{x}_t\|_2^2 + 100\sum_{t=1}^{T} I_{obstacle}(\mathbf{x}_T)\right]$$

$$\text{Where the function } I_{obstacle}(\mathbf{x}_T) = \begin{cases} 1, \text{if } \mathbf{x}_T \text{ is inside obstacle} \\ 0, \text{otherwise} \end{cases}$$

Hints:

i.) The obstacles are spherical, so to check if an **x** is inside an obstacle check the distance to the centers

ii.) The start, goal, obstacle centers and radii can be accessed from `GATrajectoryOptimizerSolution` via `self.start`, `self.goal`, `self.obstacle_centre`, `self.obstacle_radii`

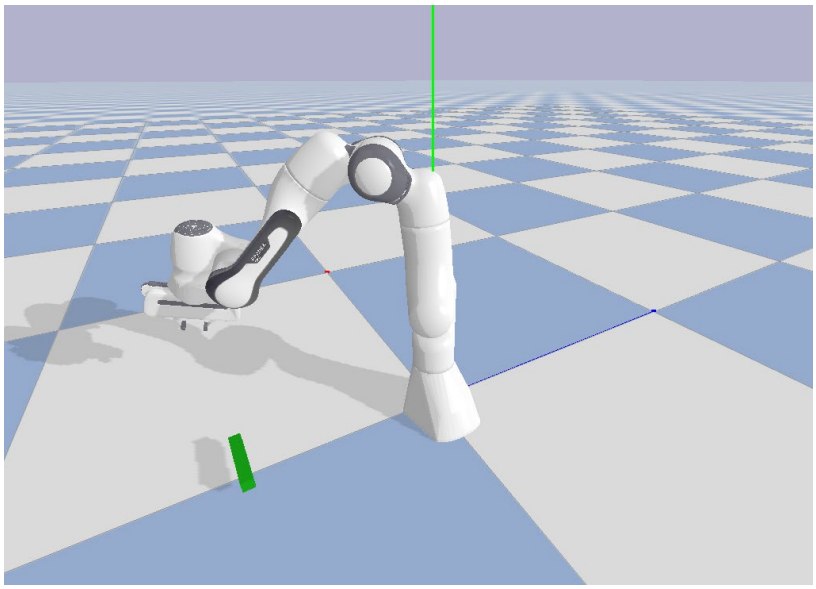
iii.) Some useful functions you might consider using: `numpy.cumsum`, `numpy.where`, `numpy.linalg.norm`

c. Implement `crossover` in `GATrajectoryOptimizerSolution` . This will take two parents and a crossover points in time `K`. You should perform the crossover in time, i.e., for parents $(\Delta\mathbf{x}_1, \Delta\mathbf{x}_2, \ldots \Delta\mathbf{x}_T)$ and $(\Delta\mathbf{y}_1, \Delta\mathbf{y}_2, \ldots \Delta\mathbf{y}_T)$ should produce children:

$$(\Delta\mathbf{x}_1, \ldots, \Delta\mathbf{x}_K, \Delta\mathbf{y}_{K+1}, \ldots, \Delta\mathbf{y}_T) \text{ and}$$
$$(\Delta\mathbf{y}_1, \ldots, \Delta\mathbf{y}_K, \Delta\mathbf{x}_{K+1}, \ldots, \Delta\mathbf{x}_T)$$

d. Implement `select_children` in `GATrajectoryOptimizerSolution` . This will take a parent and a child and return which of these should be maintained in the population, as well as its fitness. Your function should evaluate the fitness of both the child and the parent, and with probability `self.params['select_best_child_p']` it should return the one with the best fitness.
Hint: Look at `tournament_selection` in `GATrajectoryOptimizer` in `hw7/ga_trajectory_optimization/ga_trajopt.py` for a hint about how to choose an outcome with a given probability.

3. **Robot manipulation in PyBullet** [26 points]:
In this problem we will use the PyBullet simulator to simulate a robot manipulator and perform a pick-and-place task. Once you have successfully implemented the components of this question, the robot should move to pick up the block and then move and place the block in a specified goal location. **Note:** the simulation uses the convention that the y direction is upwards.



Setup:
You need to install some python libraries first to get this to run. Run the following command in your terminal: `pip3 install scipy lcm`
If you are attempting this question before the pybullet lab, you will also need to install pybullet with: `pip3 install pybullet`
You also need to generate the required LCM messages. Run the following command in your terminal from the `hw7/block_pick_and_place` directory: `lcm-gen -p block_goal.lcm`

The code consists of four python files, two of which you should modify:

- `block_pick_and_place.py` This file contains the base class `PandaSim` which contains functions for initializing the simulation, stepping the simulation, and various getter & setter methods.
- `transform_utils.py` This file contains functions for transforming between different rotation representations. You should not need to directly use anything in this file beyond the code that is already given to you.
- `block_pick_and_place_soln.py` This file contains the class `PandaSimSolution` which inherits from `PandaSim` and is where most of your code will be added. **You should modify this**.
- `block_goal_publisher.py` This file will read the goal pose of the block from a file and use LCM to publish it, you will need to implement the publishing. **You should modify this**. The subscriber is already written in `PandaSim` in `block_pick_and_place.py`.

In addition you should notice the following folders & files:

- `inputs1.txt` this is the file that contains the object goal pose. The first line is a translation, and the 2nd-4th lines are a 3D rotation matrix.
- `block_goal.lcm` this is the definition of the LCM message for publishing the goal pose of the block.
- `assets` this is a folder containing simulation assets - you do not need to look at this.

To test your code you can run `block_pick_and_place_tests.py`

Note: You need to run all of the scripts from the `hw7/block_pick_and_place` directory. Use the `cd` terminal command in the VSCode terminal to navigate to the correct directory.

Complete this problem in the following steps:

a. *Read through the code we gave you* to try to understand which functions have already been written, and what member variables you will need to read/write. This will make the rest of the problem much quicker and prevent you from accidentally re-implementing things we've already written. **The ability to read through someone else's code and get a general understanding of what it does/should do is an essential skill!** If you get confused, try making a diagram on paper that has every class (along with member functions) and the inheritance relationship between classes, then go through the code and see which file contains which function.

b. When grasping an object, we do not want to make the gripper pose exactly the same as the object pose, as they will collide, and the object may move in an undesirable way. Instead, it is common to move to a *pre-grasp* pose, which is offset from the object, and then perform a grasp. Implement the function `block_to_gripper_pose` in `block_pick_and_place_soln.py`. This function will take the `block_pose` as input, and output an offset `gripper_pose` such that the gripper and obstacle do not collide. Both `gripper_pose` and `block_pose` should be homogeneous transform matrices. The transformation from the block pose to the gripper pose is represented by the following rotation and translation **with respect to the world frame:**

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}, T = \begin{bmatrix} 0 \\ 0.015 \\ 0 \end{bmatrix}$$

c. Complete the file `block_goal_publisher.py.` There are two incomplete functions. The first is `read_goal`. This should read a file and return a homogeneous transform matrix. The second is `publish_transformation` , which should publish this matrix via LCM. `PandaSim` has a subscriber to the message and should store the result in `self.goal` , which you can use to check if your publisher is working. To check which channel you should publish to, look at `setup_thread` in `PandaSim` in `block_pick_and_place.py`. You will need to run `block_goal_publisher.py` and `block_pick_and_place_soln.py` in different terminals at the same time to test your code.

d. Now we can compute the desired end-effector pose for our robot manipulator. However, in order to reach this end effector pose we need to find the corresponding joint angles `q` so we can move the robot to that configuration. This is called **Inverse Kinematics.** We will perform **Jacobian-based Inverse Kinematics.**

   The Jacobian J describes the relation $J\dot{q} = v$, where $\dot{q}$ are the joint velocities and v is the end-effector velocity. We can use this relation to get a finite difference equation $J\Delta q = \Delta\mathbf{p}$ where $\Delta\mathbf{p}$ is the desired change in end effector pose. There are many different approaches for using this equation to estimate the change in joint configurations, but we

will use the **Jacobian pseudoinverse** method. In this method, we compute the change in joint configuration using the following equation:

$$\Delta q = J^T (JJ^T)^{-1} \Delta \mathbf{p}$$

You will now complete the missing part of `jacobian_ik` in `PandaSimSolution` in `block_pick_and_place_soln.py`. Use the above equation to compute the required change in joint configuration, storing it in the variable `dq`.
`jacobian_ik` already computes the Jacobian `J` and the desired change in end effector pose as `dp`.
You can test this function (alongside parts b,c) by running `block_pick_and_place_tests.py`.

Note: Once you have this running you may see the robot moving very quickly in the simulator - this is the expected behavior. To compute the forward kinematics and Jacobian the simulator has to 'teleport' the robot to the query configuration, and then teleport back. Don't worry about this!