Assignment 4

Assignment Objective

The objective of this assignment is to implement Dynamical Motion Primitives with obstacle avoidance for a simplified robotic setup. The central idea is to design a control law that pulls the robot to the goal while pushing it away from the obstacles.

Instructions

In this assignment, we will ask you to fill out missing pieces of code in a Python script. You will submit the completed code to autograder. Your code is run via the autograder that will verify the correctness of your work and assign you a score out of 100.

- Download the assignment 4 files from the Files menu of Canvas.
- For each "Part" of the HW assignment, fill in the missing code as described in the problem statement.
- Submit the completed `assignment_4.py` to autograder.
- The assignment is due on Nov 11^{th}, 2024.
- You will need `numpy` and `pybullet` libraries to simulate the robot.

Part 1 – Obstacle Free DMPs (30 points)

Consider the scene depicted in Fig. 1. The robot (visualized as the pink cylinder) is free to move in the (x, y) plane. The objective here is to write a control law such that for any random initialization, the robot always travels to its goal configuration at $(0, 0)$. In the following, we denote the configuration and velocity of the robot with $\boldsymbol{q}$ and $\boldsymbol{v}$ respectively.

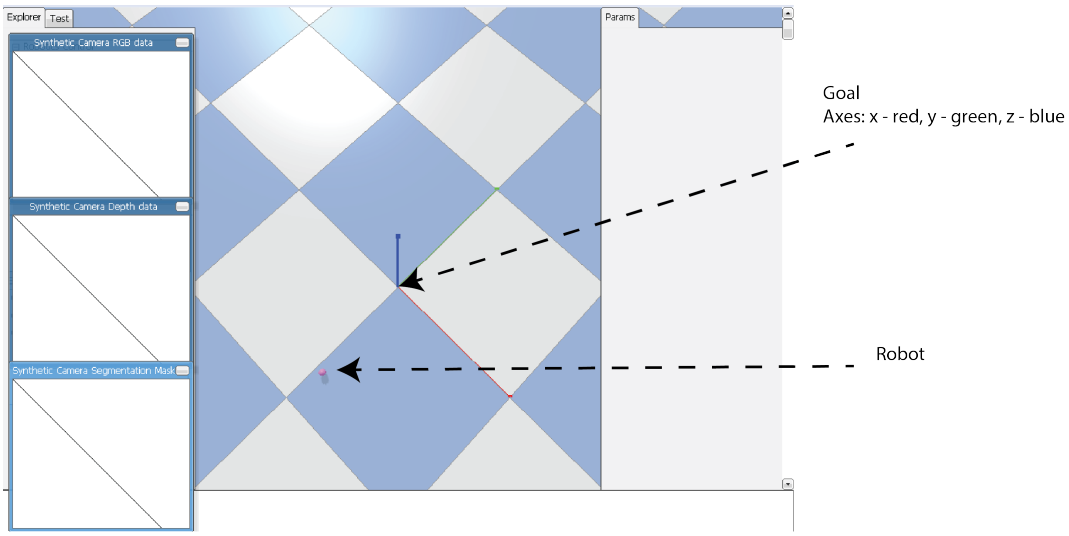

Figure 1. The point robot and scene. The robot is visualized with the pink cylinder and moves in the (x, y) plane.

Recall that the governing equations of motion for the robot are:

$$u_x = ma_x$$
$$u_y = ma_y$$

where u_x denotes the force applied along the x axis and a_x denotes the resulting acceleration. The mass of the robot is denoted by m. Implement a Dynamical Motion Primitive (DMP) controller to drive the robot configuration $(x, y)_t$ to the goal configuration $(0, 0)$ as $t \to \infty$. Recall from lecture that the DMP controller drives the system to behave as though a virtual spring and damper are placed between the robot and the goal where the set point of the spring is at the goal.

To this end, complete the `dmp_control` function in `assignment_6.py` where the inputs to the function are the current configuration `q` as a `np.array((2, ))` and the current velocity `v` as a `np.array((2, ))`. The output of the function is the control `u` as a `numpy.array((2, ))` where the first element is u_x and the second is u_y. In case you need it, the inertia of the robot is defined as `ROBOT_MASS`. You are free to choose your spring and damper constants; however, be sure that the robot reaches the goal (or is within 1 cm) before the end of the simulation time. To test your code, run the `main` function with the argument `with_obs = False`. You'll see a visualization of the robot motion under the control law you provide.

Part 2 – DMPs to Avoid Obstacles (70 points)

In addition to trajectory tracking, DMPs can also be used for collision avoidance. How you may ask? Remember that the control law we defined for a DMP places a global attractor at the goal configuration. From any initial configuration, the attractor pulls the robot to the goal. We can similarly design repellers which push the robot away from regions in the state space. By placing repellers over obstacles, we can make sure that the robot avoids them while traveling to the goal. This idea has been around for a while under various guises, the most famous of which is "potential fields" – check this rich literature out if you're interested in more details.

In this part, we're going to implement a fairly simple version of repellers. To start, consider the left panel of Fig. 2 in which 2 randomly located but fixed size obstacles (visualized in cyan) are introduced into the scene. Our goal is to design a control law that pulls the robot to the same goal configuration as the previous part $(x, y) = (0, 0)$ while dodging the obstacles. Our implementation follows the work of "Biologically-inspired dynamical systems for movement generation: automatic real-time goal adaptation and obstacle avoidance," H. Hoffmann, P. Pastor, D. H. Park, and S. Schaal, ICRA 2009.

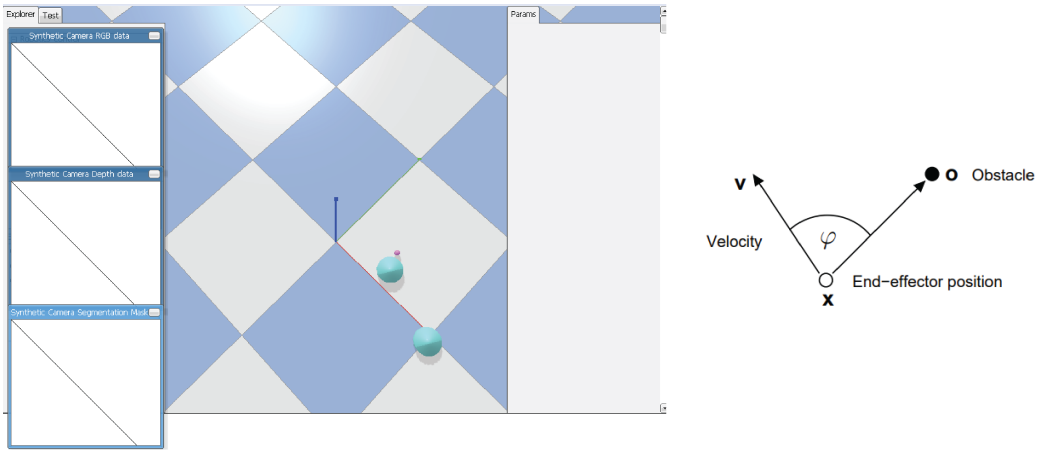

Figure 2. Left panel: Robot (in pink) and obstacles (in Cyan). Right panel: Visualization of the definition of the steering angle.

The first thing we need to do is to define the steering angle. A visualization of the steering angle is show in the right panel of Fig. 2. The steering angle is an indicator of possible collisions, low angles would imply we're heading to the obstacles and high angles would imply we're clear. A reasonable equation to describe the *change* in steering angle to avoid collisions would be:

$$\dot{\varphi} = \gamma\varphi \exp\left(-\beta|\varphi|\right)$$

where for low values of the steering angle we'd have large changes in the angle (we'd like to rapidly change the angle if we're moving towards the obstacle) and we'd like to not change the angle if we're not heading

towards an obstacle. A great way to verify this is to plot this function with the x axis as φ and the y axis as $\dot{\varphi}$. This plot is typically referred to as a phase plot (Fig. 5 of the paper). We do not expect this as part of the assignment.

Next, we need to combine the equation describing the change in steering angle with the velocity of the robot. One effective approach is to relate the change in steering angle to the change in velocity of the robot is to write:

$$\dot{\boldsymbol{v}} = \mathbf{R}\boldsymbol{v}\dot{\varphi}$$

where $\mathbf{R}$ is a rotation matrix with axis $\boldsymbol{r} = (\boldsymbol{o} - \boldsymbol{x}) \times \boldsymbol{v}$ with angle of rotation $\pi/2$. The intuition behind this expression is that we'd like the velocity of the robot to change orthogonal to it's current heading proportional to the rate of change in the steering angle. The smaller the steering angle, the larger we need to make a course correction perpendicular to our current heading. We can replace this expression into our DMP controller from the previous section as:

$$\boldsymbol{u} = \boldsymbol{u}_{\text{DMP}} + \gamma \mathbf{R}\boldsymbol{v}\varphi \exp\left(-\beta\varphi\right) = \boldsymbol{u}_{\text{DMP}} + \boldsymbol{u}_{\text{coll\_avoid}}$$

with:

$$\varphi = \cos^{-1}\left(\frac{(\boldsymbol{o} - \boldsymbol{x})^T(\boldsymbol{v})}{|\boldsymbol{o} - \boldsymbol{x}| \times |\boldsymbol{v}|}\right)$$

and note that φ is always positive from this calculation – this is desired. The notation $|.|$ denotes the 2-norm (Euclidean length) of the vectors. The rotation matrix $\mathbf{R}$ determines the direction in which the velocity is to be changed, the remaining terms determine the magnitude of change. Implement this controller in `collision_avoidance.py`. This function takes `q` and `v` of type `np.array((2, ))` as input and returns `u_coll_avoid` of type `np.array((2, ))` as output. Note that the positions of the obstacles are given by a global variable `OBS` in the beginning of the code. The simulation code takes care of adding the DMP control with the collision avoidance control. You'll need to tune γ and β, keep in mind they are positive. A good starting point is $\gamma = 150$ and $\beta = 3.5$; however, you may need to adjust these values depending on your implementation. To test your code, run the `main` function with the argument `with_obs = True`. You'll see a visualization of the robot motion under the control law you provide. You'll be graded on your robots ability to reach the goal without hitting anything.