CS287-H AHRI HW1

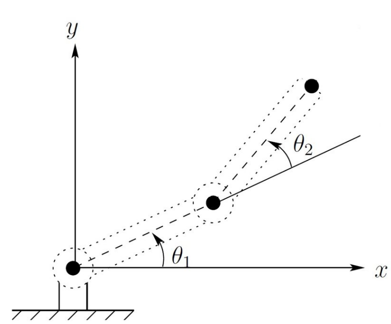
In HW1 we will walk through an implementation of trajectory optimization using CHOMP.

First, let's install and import the dependencies for this notebook:

In [1]:
```
%pip install numpy matplotlib scipy ipympl git+https://github.com/fogleman/sdf
%matplotlib ipympl

import random
import time
import math
import numpy as np
from scipy import stats
from scipy.ndimage.morphology import distance_transform_edt
import scipy.spatial
import matplotlib.pyplot as plt
from IPython import display
from matplotlib import animation
```

```
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.8/dist-packages (from nbconvert->notebook>=
4.4.1->widgetsnbextension~=3.6.0->ipywidgets<9,>=7.6.0->ipympl) (4.6.3)
Requirement already satisfied: defusedxml in /usr/local/lib/python3.8/dist-packages (from nbconvert->notebook>=4.4.1
->widgetsnbextension~=3.6.0->ipywidgets<9,>=7.6.0->ipympl) (0.7.1)
Requirement already satisfied: fastjsonschema in /usr/local/lib/python3.8/dist-packages (from nbformat->notebook>=4.
4.1->widgetsnbextension~=3.6.0->ipywidgets<9,>=7.6.0->ipympl) (2.16.3)
Requirement already satisfied: jsonschema>=2.6 in /usr/local/lib/python3.8/dist-packages (from nbformat->notebook>=
4.4.1->widgetsnbextension~=3.6.0->ipywidgets<9,>=7.6.0->ipympl) (4.3.3)
Requirement already satisfied: pyrsistent!=0.17.0,!=0.17.1,!=0.17.2,>=0.14.0 in /usr/local/lib/python3.8/dist-packag
es (from jsonschema>=2.6->nbformat->notebook>=4.4.1->widgetsnbextension~=3.6.0->ipywidgets<9,>=7.6.0->ipympl) (0.19.
3)
Requirement already satisfied: attrs>=17.4.0 in /usr/local/lib/python3.8/dist-packages (from jsonschema>=2.6->nbform
at->notebook>=4.4.1->widgetsnbextension~=3.6.0->ipywidgets<9,>=7.6.0->ipympl) (22.2.0)
Requirement already satisfied: importlib-resources>=1.4.0 in /usr/local/lib/python3.8/dist-packages (from jsonschema
>=2.6->nbformat->notebook>=4.4.1->widgetsnbextension~=3.6.0->ipywidgets<9,>=7.6.0->ipympl) (5.12.0)
Requirement already satisfied: cffi>=1.0.1 in /usr/local/lib/python3.8/dist-packages (from argon2-cffi-bindings->arg
on2-cffi->notebook>=4.4.1->widgetsnbextension~=3.6.0->ipywidgets<9,>=7.6.0->ipympl) (1.15.1)
Requirement already satisfied: webencodings in /usr/local/lib/python3.8/dist-packages (from bleach->nbconvert->noteb
ook>=4.4.1->widgetsnbextension~=3.6.0->ipywidgets<9,>=7.6.0->ipympl) (0.5.1)
Requirement already satisfied: pycparser in /usr/local/lib/python3.8/dist-packages (from cffi>=1.0.1->argon2-cffi-bi
```

We will be generating trajectories for a simple 2-dimensional robot arm with two degrees of freedom, represented by angle joints θ_1 and θ_2:



Functions which implement the kinematics and inverse kinematics for this robot are given below. Recall that forward kinematics maps a point in configuration space $q = (\theta_1, \theta_2)$ to a point in world space (x, y). Inverse kinematics gives a set of possible points in configuration space which could map to a given point in world space.

We will assume that the only part of the robot which can collide with obstacles is the end effector at the point (x, y) in world space.

In [2]:

```
def forward_kinematics(theta1, theta2):
    return (
        np.cos(theta1) + np.cos(theta1 + theta2),
        np.sin(theta1) + np.sin(theta1 + theta2),
    )

def inverse_kinematics(x, y):
    qs = []
    r2 = x ** 2 + y ** 2
    theta1 = np.arctan2(
        y * r2 + x * np.sqrt((4 - r2) * r2),
        x * r2 - y * np.sqrt((4 - r2) * r2),
    )
    theta2 = np.arctan2(
        -np.sqrt((4 - r2) * r2),
        r2 - 2,
    )
    qs.append((theta1, theta2))
    theta1 = np.arctan2(
        y * r2 - x * np.sqrt((4 - r2) * r2),
        x * r2 + y * np.sqrt((4 - r2) * r2),
    )
    theta2 = np.arctan2(
        np.sqrt((4 - r2) * r2),
        r2 - 2,
    )
    qs.append((theta1, theta2))

    for theta1, theta2 in list(qs):
        for delta_theta1 in [-2 * np.pi, 0, 2 * np.pi]:
            for delta_theta2 in [-2 * np.pi, 0, 2 * np.pi]:
                if (
                    not (delta_theta1 == 0 and delta_theta2 == 0)
                    and -2 * np.pi <= theta1 + delta_theta1 <= 2 * np.pi
                    and -2 * np.pi <= theta2 + delta_theta2 <= 2 * np.pi
                ):
                    qs.append((theta1 + delta_theta1, theta2 + delta_theta2))

    return qs
```

Constructing the cost function

To start, we'll construct the cost function used by CHOMP. The obstacle cost function is based on the *signed distance* to the nearest object. To simply the calculation of the signed distance, we'll use the `sdf` library. Throughout the rest of the assignment, we'll work with a grid of evenly spaced points in both W-space and C-space.

In [3]:

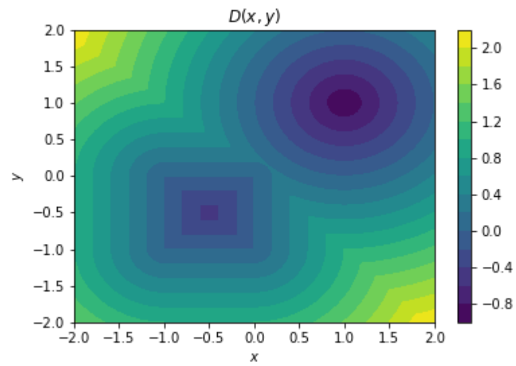
```
# Grid for W-space with resolution of 0.02
xs, ys = np.linspace(-2, 2, 201), np.linspace(-2, 2, 201)
x_grid, y_grid = np.meshgrid(xs, ys, indexing="ij")

# Grid for C-space with a resolution of pi / 50
theta1s, theta2s = np.linspace(-2 * np.pi, 2 * np.pi, 201), np.linspace(-2 * np.pi, 2 * np.pi, 201)
theta1_grid, theta2_grid = np.meshgrid(theta1s, theta2s, indexing="ij")

def get_W_signed_distance(obstacles, x_grid=x_grid, y_grid=y_grid):
    flat_signed_distances = obstacles(np.stack([x_grid.flat, y_grid.flat], axis=1))
    #print(flat_signed_distances)
    return flat_signed_distances.reshape(x_grid.shape)
```

In [3]:

For instance, if we have a circular and a recangular obstacle, the signed distance function in W-space will look like this:

In [4]:
```python
from sdf.d2 import rectangle, circle

obstacles = rectangle(a=(-1, -1), b=(0, 0)) | circle(radius=1, center=(1, 1))
signed_distance = get_W_signed_distance(obstacles)

plt.figure()
plt.contourf(x_grid, y_grid, signed_distance, levels=20)
plt.colorbar()
plt.title("$D(x, y)$")
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.show()
```

Figure



In [5]:
```python
from google.colab import output
output.enable_custom_widget_manager()
```

You should now implement the following function which calculates the cost function for CHOMP in C-space. The cost function is given by

$$c(q) = \begin{cases} -D(q) + \epsilon/2 & D(q) \leq 0 \\ (D(q) - \epsilon)^2/(2\epsilon) & 0 < D(q) \leq \epsilon \\ 0 & \text{otherwise} \end{cases}$$

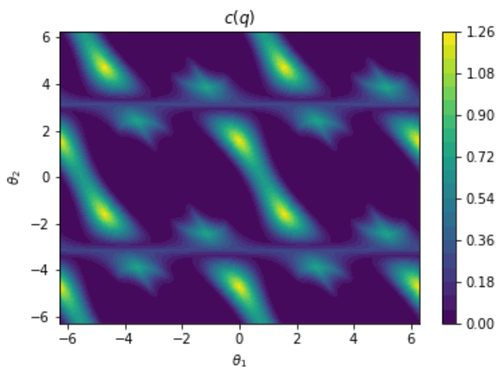
```
# This is formatted as code
```

In [6]:
```python
def get_obs_cost(obstacles, eps=0.5):
    """
    Given a set of obstacles from the sdf library, this function should return a grid
    of shape (len(theta1s), len(theta2s)) with the obstacle cost for each point in
    C-space.
    """
    obs_cost = np.zeros_like(theta1_grid)
    def helper(dist):
      if dist <= 0:
        return -dist + eps/2
      if dist <= eps:
        return (dist-eps)**2 / (2*eps)
      return 0

    for i in range(201):
      for j in range(201):
        theta1, theta2 = theta1_grid[i][j], theta2_grid[i][j]
        x, y = forward_kinematics(theta1, theta2)
        dist = get_W_signed_distance(obstacles, x, y)
        cq = helper(dist)
        obs_cost[i][j] = cq

    return obs_cost
```

To test your function, we can look at what the obstacle cost function looks like for the example above:

In [7]:
```
obs_cost = get_obs_cost(obstacles)

plt.figure()
plt.contourf(theta1_grid, theta2_grid, obs_cost, levels=20)
plt.title("$c(q)$")
plt.xlabel(r"$\theta_1$")
plt.ylabel(r"$\theta_2$")
plt.colorbar()
plt.show()
```

Figure



In [8]:
```
obs_cost[0][0]
```

Out[8]:
```
0.007359312880714951
```

Calculating the gradient of the cost function

To implement functional gradient descent, we also need to know the gradient of the cost function. Rather than calculating an explicit formula for the gradient, we'll use the method of *finite differences*:

$$\frac{\partial c(\theta_1, \theta_2)}{\partial \theta_1} \approx \frac{c(\theta_1 + h, \theta_2) - c(\theta_1, \theta_2)}{h}$$

In this case, we'll just let $h = \pi/50$, i.e., the resolution of the grid. Implement the below function to calculate the gradient of the cost function.

In [9]:
```
def get_cost_grad(cost, h=np.pi/50):
    """
    Given a cost function grid of shape (len(theta1s), len(theta2s)), return a tuple
    (cost_grad_theta1, cost_grad_theta2) of the gradient with respect to theta1
    and theta2, where cost_grad_theta1.shape == cost_grad_theta2.shape == cost.shape.
    """
    # Shift cost values by h to the left and to the right along theta1 axis
    right_cost = np.roll(cost, -1, axis=0)

    # Calculate the gradient of the cost function along theta1 axis
    cost_grad_theta1 = (right_cost - cost) / (h)

    # Shift cost values by h to the left and to the right along theta2 axis
    down_cost = np.roll(cost, -1, axis=1)

    # Calculate the gradient of the cost function along theta2 axis
    cost_grad_theta2 = (down_cost - cost) / (h)

    return cost_grad_theta1, cost_grad_theta2
```

In [10]:
```
obs_cost_grad_theta1, obs_cost_grad_theta2 = get_cost_grad(obs_cost)
assert (
    obs_cost_grad_theta1.shape == obs_cost.shape
    and obs_cost_grad_theta2.shape == obs_cost.shape
)
```

Optimizing the cost function

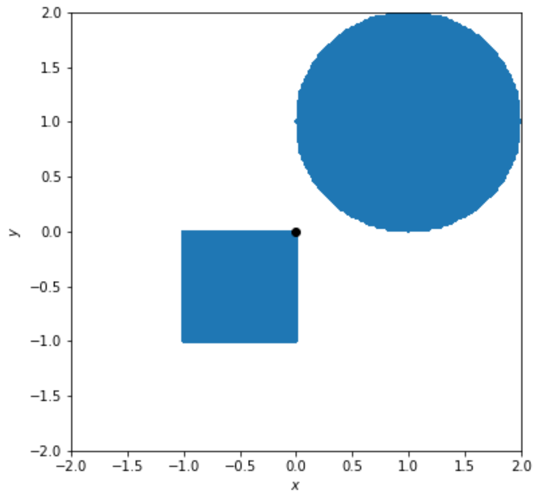
Now we are almost ready to optimize a trajectory with respect to our cost function! To start, we'll define a few helper functions:

In [11]:
```python
def get_straight_line_traj(q_start, q_goal, num_waypoints = 30):
    q_start = np.array(q_start)
    q_goal = np.array(q_goal)
    alpha = np.linspace(0, 1, num_waypoints)
    return (1 - alpha[:, None]) * q_start[None, :] + alpha[:, None] * q_goal[None, :]

def compute_traj_cost(traj, obs_cost, smoothness_weight):
    traj_obs_cost = 0
    traj_smoothness_cost = 0

    for t, (theta1, theta2) in enumerate(traj):
        traj_obs_cost += obs_cost[
            int(theta1 * 50 / np.pi) + 100,
            int(theta2 * 50 / np.pi) + 100,
        ]
        if t < traj.shape[0] - 1:
            traj_smoothness_cost += np.linalg.norm(traj[t + 1] - traj[t]) ** 2

    return traj_obs_cost + smoothness_weight * traj_smoothness_cost

def play_traj(traj, obstacles, resolution=5):
    obstacle_image = get_W_signed_distance(obstacles) <= 0
    fig, ax = plt.subplots(figsize=(6, 6))
    ax.contourf(x_grid, y_grid, obstacle_image, levels=1, colors=["w", "C0"])
    ax.set_xlabel("$x$")
    ax.set_ylabel("$y$")
    line, = ax.plot([0, 0, 0], [0, 0, 0], c="k", markevery=[2], marker="o")
    def animate(step):
        t = step // resolution
        theta1_a, theta2_a = traj[t]
        theta1_b, theta2_b = traj[min(len(traj) - 1, t + 1)]
        substep = (step % resolution) / resolution
        theta1 = (1 - substep) * theta1_a + substep * theta1_b
        theta2 = (1 - substep) * theta2_a + substep * theta2_b

        line.set_xdata([0, np.cos(theta1), np.cos(theta1) + np.cos(theta1 + theta2)])
        line.set_ydata([0, np.sin(theta1), np.sin(theta1) + np.sin(theta1 + theta2)])
        return line,
    anim = animation.FuncAnimation(
        fig,
        animate,
        range(len(traj) * resolution),
        interval=100 // resolution,
        repeat=False,
        blit=True,
    )
    plt.show()
    return anim
```

We'll start our optimization from a trajectory which forms a straight line in C-space. Let's use some of our helper functions to see what such a trajectory looks like:

In [12]:
```python
traj = get_straight_line_traj((0, 0), (-1.3 * np.pi, 2 * np.pi))
play_traj(traj, obstacles)
```

Figure



Out[12]: `<matplotlib.animation.FuncAnimation at 0x7f5e82bee940>`

In [13]:
```python
traj[0]
```

Out[13]: `array([0., 0.])`

The function `compute_traj_cost` that is implemented above computes the total cost for a trajectory, which is the sum of the obstacle cost and a smoothness cost weighted by the smoothness weight λ:

$$\mathcal{U}[\zeta] = \mathcal{U}_{\text{obs}}[\zeta] + \lambda\, \mathcal{U}_{\text{smooth}}[\zeta].$$

The obstacle cost is simply the sum of the obstacle costs over the trajectory:

$$\mathcal{U}_{\text{obs}}[\zeta] = \sum_{t=1}^{T} c(\zeta(t)).$$

Meanwhile, the smoothness cost encourages the trajectory to keep an approximately constant velocity:

$$\mathcal{U}_{\text{smooth}}[\zeta] = \sum_{t=1}^{T-1} \|\zeta(t+1) - \zeta(t)\|^2.$$

Based off of `compute_traj_cost` , implement the following function to calculate the functional gradient of the cost $\mathcal{U}[\zeta]$. The functional gradient has the form

$$\nabla \mathcal{U}[\zeta](t) = \nabla c(\zeta(t)) + \lambda(2\zeta(t) - \zeta(t+1) - \zeta(t-1)).$$

In [14]:
```python
def compute_traj_cost_grad(traj, obs_cost, smoothness_weight):
    # Compute gradient of obstacle cost
    obs_cost_grad_theta1, obs_cost_grad_theta2 = get_cost_grad(obs_cost)

    # Initialize gradient array
    traj_grad = np.zeros_like(traj)

    for t, (theta1, theta2) in enumerate(traj):
        theta1_grad = obs_cost_grad_theta1[
            int(theta1 * 50 / np.pi) + 100,
            int(theta2 * 50 / np.pi) + 100]
        theta2_grad = obs_cost_grad_theta2[
            int(theta1 * 50 / np.pi) + 100,
            int(theta2 * 50 / np.pi) + 100]
        traj_grad[t] = np.array([theta1_grad, theta2_grad])
        if t >= 1 and t < traj.shape[0]-1:
            traj_grad[t] += smoothness_weight * (2 * traj[t] - traj[t+1] - traj[t-1])

    return traj_grad
    # YOUR SOLUTION HERE
```

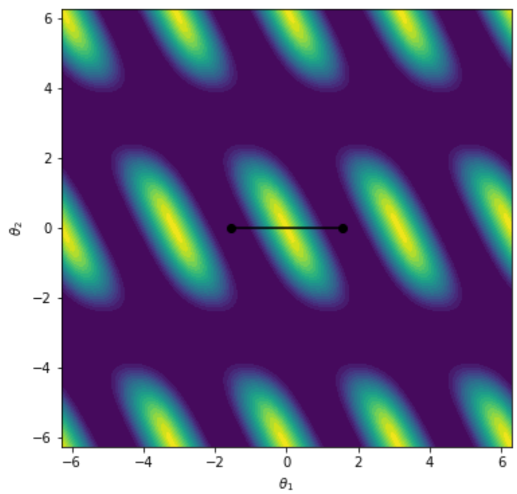
Now we are ready to run CHOMP! We've provided the rest of the implementation below.

In [15]:
```python
def run_chomp(
    traj,
    obs_cost,
    smoothness_weight,
    num_iterations,
    learning_rate,
    use_covariant_gradient=False,
):
    num_waypoints, _ = traj.shape

    A = np.zeros((num_waypoints, num_waypoints))
    for i in range(num_waypoints):
        A[i, i] = 2
        if i >= 1:
            A[i, i - 1] = -1
        if i < num_waypoints - 1:
            A[i, i + 1] = -1
    Aiv = np.linalg.inv(A)

    traj_progress = np.empty((num_iterations + 1,) + traj.shape)
    traj_progress[0] = traj
    for it in range(num_iterations):
        traj_grad = compute_traj_cost_grad(traj, obs_cost, smoothness_weight)

        if use_covariant_gradient:
            traj_grad_theta1 = traj_grad[:, 0]
            traj_grad_theta2 = traj_grad[:, 1]
            traj_grad_theta1_smooth = Aiv @ traj_grad_theta1
            traj_grad_theta2_smooth = Aiv @ traj_grad_theta2
            traj_grad_smooth = np.stack([traj_grad_theta1_smooth, traj_grad_theta2_smooth], axis=1)
            traj_grad = traj_grad_smooth

        traj_inc = -learning_rate * traj_grad
        # Don't change the start and end points.
        traj_inc[0, :] = 0
        traj_inc[-1, :] = 0
        traj = traj + traj_inc
        traj = traj.clip(-2 * np.pi, 2 * np.pi)

        traj_progress[it + 1] = traj

    final_cost = compute_traj_cost(traj, obs_cost, smoothness_weight)

    return traj_progress, final_cost


def visualize_traj_progress(traj_progress, obs_cost):
    num_iterations, num_waypoints, _ = traj_progress.shape
    fig, ax = plt.subplots(figsize=(6, 6))
    ax.contourf(theta1_grid, theta2_grid, obs_cost, levels=20)
    ax.set_xlabel(r"$\theta_1$")
    ax.set_ylabel(r"$\theta_2$")
    line, = ax.plot(
        traj_progress[0, :, 0],
        traj_progress[0, :, 1],
        c="k", markevery=[0, num_waypoints - 1], marker="o",
    )
    def animate(it):
        line.set_xdata(traj_progress[it, :, 0])
        line.set_ydata(traj_progress[it, :, 1])
        return line,
    anim = animation.FuncAnimation(
        fig,
        animate,
        range(0, num_iterations, max(1, num_iterations // 50)),
        interval=100,
        repeat=False,
        blit=True,
    )
    plt.show()
    return anim
```

Let's apply CHOMP to this problem where we need to navigate the end effector through a space between two obstacles.

In [16]:
```
from sdf.d2 import circle, scale

obstacles = scale(circle(center=(-2, 0)) | circle(center=(2, 0)), (1, 2))
q_start = inverse_kinematics(0, -2)[0]
q_goal = inverse_kinematics(0, 2)[0]
obs_cost = get_obs_cost(obstacles)
initial_traj = get_straight_line_traj(q_start, q_goal)

traj_progress, final_cost = run_chomp(initial_traj, obs_cost, 1, 1000, learning_rate=0.3)
_ = visualize_traj_progress(traj_progress, obs_cost)
print("Final cost =", final_cost)
```

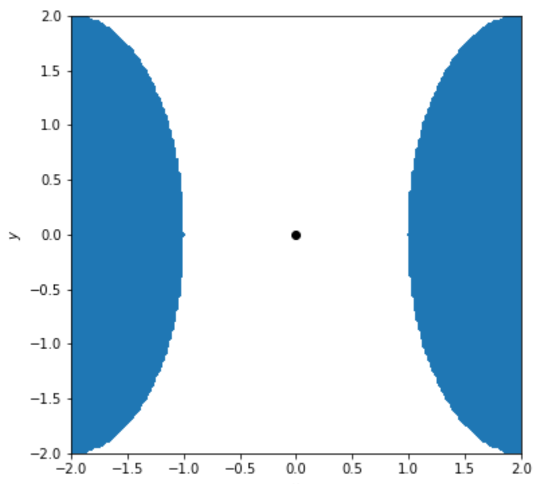
Figure




```
Final cost = 1.7962968893032178
```

In [17]:
```
# Visualize the final trajectory.
play_traj(traj_progress[-1], obstacles)
```

Figure



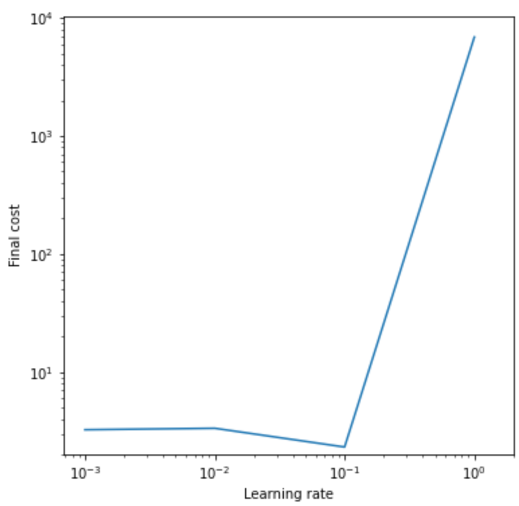
Out[17]: <matplotlib.animation.FuncAnimation at 0x7f5e82acc070>

If everything was implemented correctly, the end effector should avoid the obstacles to either side and reach the goal!

Next, let's experiment a bit with the sensitivity of our algorithm to hyperparameters. Specifically, we'll look at the learning rate. Below, make a log-log plot of the learning rate (try from 10^{-3} to 10^{0}) versus the final cost when optimizing for a solution to the above problem.

In [18]:
```python
# YOUR PLOTTING CODE HERE

learning_rates = [10**(-i) for i in range(4)]
final_costs = []

for lr in learning_rates:
    final_cost = run_chomp(initial_traj, obs_cost, 1, 1000, learning_rate=lr)[1]
    final_costs.append(final_cost)

plt.loglog(learning_rates, final_costs)
plt.xlabel('Learning rate')
plt.ylabel('Final cost')
plt.show()
```

Figure



As you may have noticed, our current implementation of CHOMP is quite sensitive to the learning rate! For many choices of learning rate, it doesn't even find a solution with no collisions. We'll fix this in the next part.

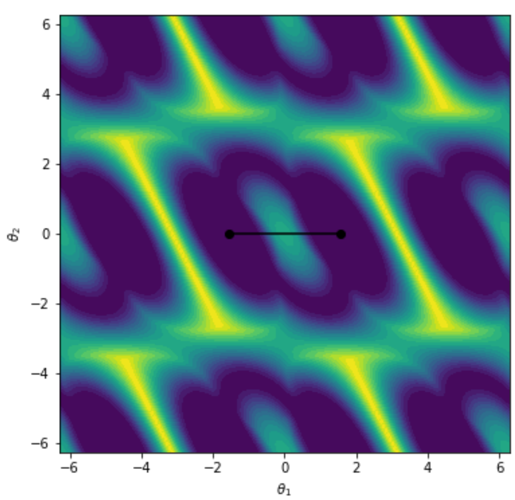
Covariant inner product for CHOMP

Rather than using the gradient directly for optimization, CHOMP first multiplies it by the inverse of the kernel matrix of the inner product corresponding to the smoothness penalty in the cost functional. We've already calculated the A matrix and its inverse in the `run_chomp` function. Modify `run_chomp` to multiply the gradient for each C-space variable by A^{-1} if `use_covariant_gradient` is `True`.

To test CHOMP with the covariant gradient, let's try a slightly harder problem with a narrower gap to fit through:

In [19]:
```
obstacles = rectangle(a=(-3, -1), b=(0.5, 1)) | rectangle(a=(1.5, -1), b=(3, 1))
q_start = inverse_kinematics(0, -2)[0]
q_goal = inverse_kinematics(0, 2)[0]
obs_cost = get_obs_cost(obstacles)
initial_traj = get_straight_line_traj(q_start, q_goal)

traj_progress, final_cost = run_chomp(
    initial_traj, obs_cost, 1, 1000, learning_rate=0.01, use_covariant_gradient=True
)
visualize_traj_progress(traj_progress, obs_cost)
```

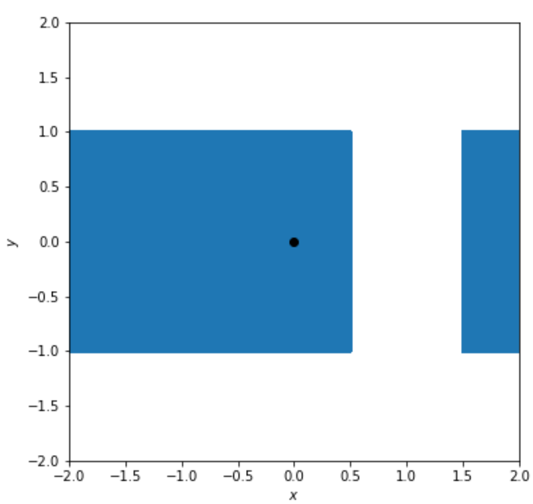
Figure



Out[19]: <matplotlib.animation.FuncAnimation at 0x7f5e828d9b20>

In [20]:
```
play_traj(traj_progress[-1], obstacles)
```

Figure

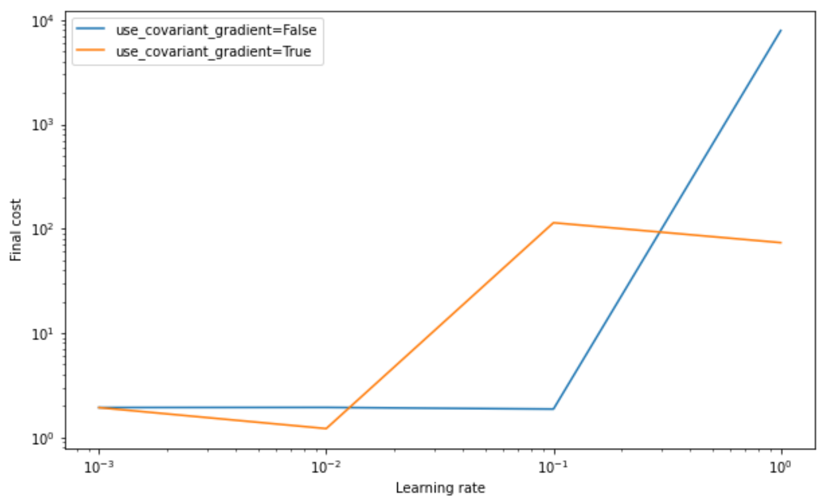


Out[20]: <matplotlib.animation.FuncAnimation at 0x7f5e82825700>

:For this optimization problem, make a plot similar to the above showing the learning rate versus the final cost. Plot two lines, one with `use_covariant_gradient=False` and one with `use_covariant_gradient=True` .

In [21]:
```
# YOUR PLOT HERE
learning_rates = [10**(-i) for i in range(4)]
final_costs_no_grad = []
final_costs_grad = []


for lr in learning_rates:
    final_cost_ng = run_chomp(initial_traj, obs_cost, 1, 1000, learning_rate=lr, use_covariant_gradient=False)[1]
    final_cost_g = run_chomp(initial_traj, obs_cost, 1, 1000, learning_rate=lr, use_covariant_gradient=True)[1]
    final_costs_no_grad.append(final_cost_ng)
    final_costs_grad.append(final_cost_g)

fig, ax = plt.subplots(figsize=(10, 6))

ax.loglog(learning_rates, final_costs_no_grad, label="use_covariant_gradient=False")
ax.loglog(learning_rates, final_costs_grad, label="use_covariant_gradient=True")

ax.set_xlabel("Learning rate")
ax.set_ylabel("Final cost")
ax.legend()

plt.show()
```

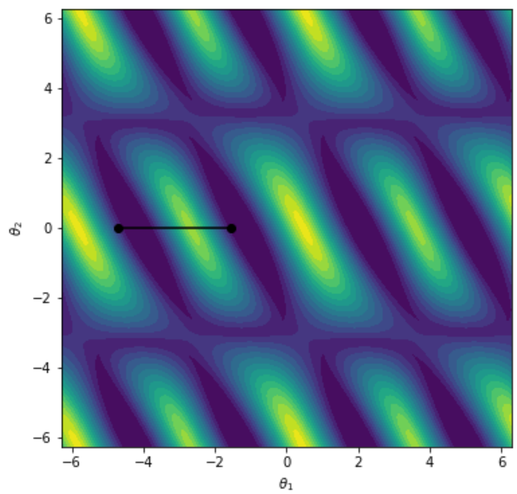
Figure



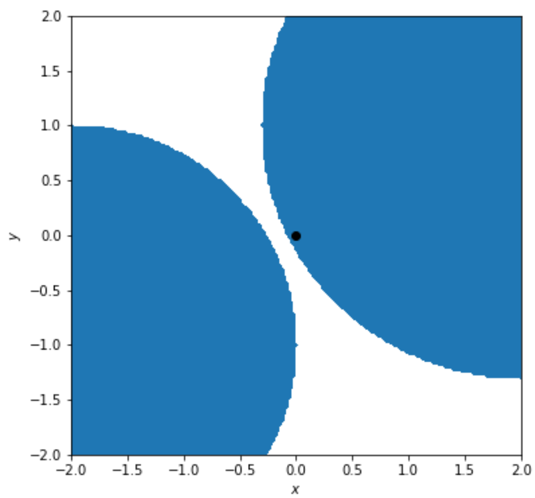
Now, can you come up with a trajectory optimization problem that the CHOMP implementation with `use_covariant_gradient=True` can't solve? Construct obstacles below along with a start and goal position such that CHOMP is unable to find a collision free path.

```
In [22]: obstacles = circle(2.3, center=(2, 1)) | circle(2, center=(-2, -1))
q_start = inverse_kinematics(0, -2)[0]
q_goal = inverse_kinematics(0, 2)[3]
obs_cost = get_obs_cost(obstacles)
initial_traj = get_straight_line_traj(q_start, q_goal)

traj_progress, final_cost = run_chomp(
    initial_traj, obs_cost, 1, 1000, learning_rate=0.01, use_covariant_gradient=True
)
anim1 = visualize_traj_progress(traj_progress, obs_cost)
anim2 = play_traj(traj_progress[-1], obstacles)
```

Figure



Figure

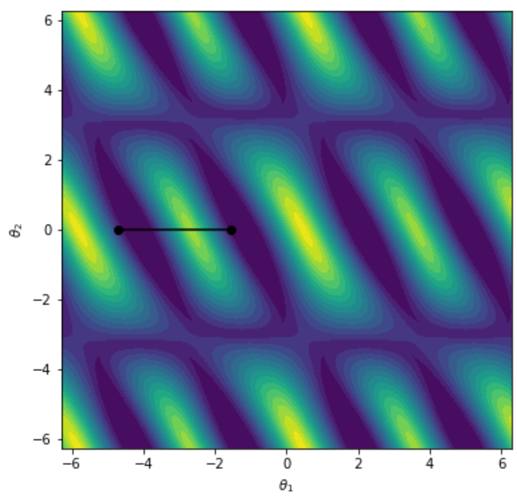


Now, how might you help CHOMP find a collision-free trajectory even in this challenging case? Write an idea below. Then, copy your trajectory optimization problem above and see if you can implement your idea to find a collision-free trajectory.

Increased the number of iterations and decrease the learning rate to avoid making big steps that may hit the obstacles.

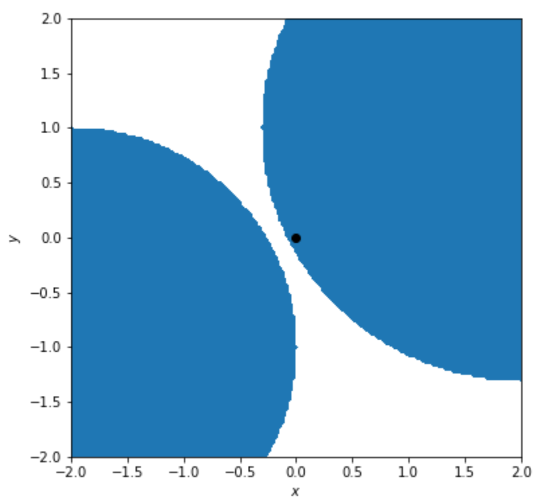
In [23]:
```python
# Copy the code above and modify it to help CHOMP find a collision-free trajectory.
obstacles = circle(2.3, center=(2, 1)) | circle(2, center=(-2, -1))
q_start = inverse_kinematics(0, -2)[0]
q_goal = inverse_kinematics(0, 2)[3]
obs_cost = get_obs_cost(obstacles)
initial_traj = get_straight_line_traj(q_start, q_goal)

traj_progress, final_cost = run_chomp(
    initial_traj, obs_cost, 1, 5000, learning_rate=0.003, use_covariant_gradient=True
)
anim1 = visualize_traj_progress(traj_progress, obs_cost)
anim2 = play_traj(traj_progress[-1], obstacles)
```

Figure




```
/usr/local/lib/python3.8/dist-packages/matplotlib/animation.py:887: UserWarning: Animation was deleted without rende
ring anything. This is most likely not intended. To prevent deletion, assign the Animation to a variable, e.g. `anim
`, that exists until you have outputted the Animation using `plt.show()` or `anim.save()`.
  warnings.warn(
```

Figure

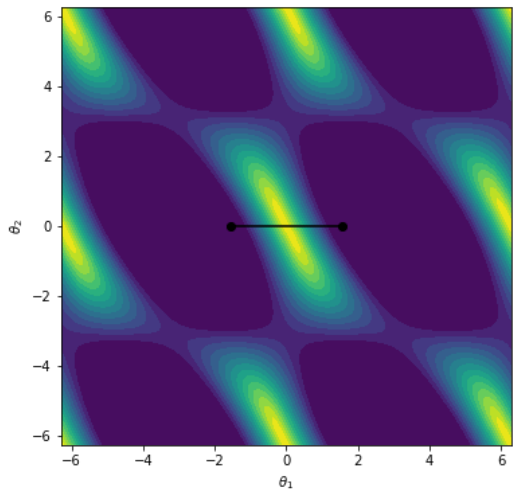


Goal sets

There is a drawback of the method we've been using so far to find a trajectory between two points in W-space. Looking back at the `inverse_kinematics` function, notice that it returns a *list* of possible C-space points for any point in W-space. If we want our robot arm to reach a certain point in W-space, then we should be able to choose any one of these C-space points as our goal endpoint for trajectory optimization. Sometimes, this can help us find a collision-free trajectory when the first goal point in C-space we try doesn't work. For instance, consider the following trajectory optimization problem:

In [24]:
```
obstacles = circle(2, center=(2, 0))
q_start = inverse_kinematics(0, -2)[0]
q_goal = inverse_kinematics(0, 2)[0]
obs_cost = get_obs_cost(obstacles)
initial_traj = get_straight_line_traj(q_start, q_goal)

traj_progress, final_cost = run_chomp(
    initial_traj, obs_cost, 1, 1000, learning_rate=0.01, use_covariant_gradient=True
)
visualize_traj_progress(traj_progress, obs_cost)
```

Figure

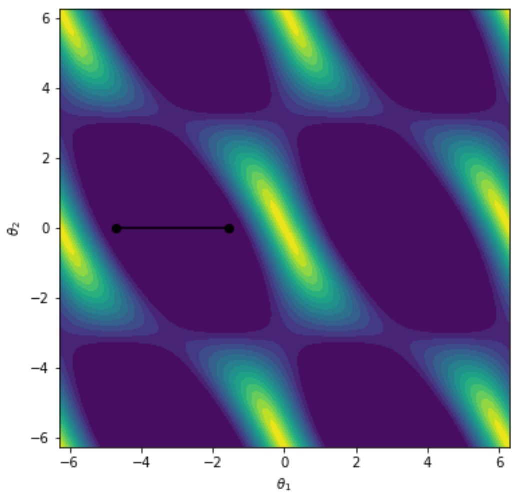


Out[24]: `<matplotlib.animation.FuncAnimation at 0x7f5e810b9910>`

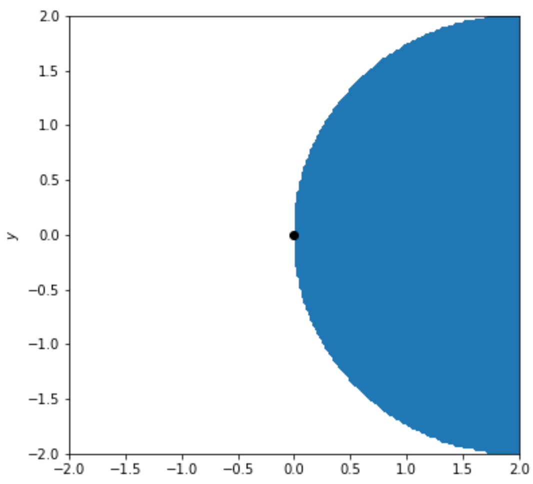
Notice that there is no collision-free path between the two points in C-space! However, if we try a different C-space point returned from `inverse_kinematics` , it works much better:

In [25]:
```
obstacles = circle(2, center=(2, 0))
q_start = inverse_kinematics(0, -2)[0]
q_goal = inverse_kinematics(0, 2)[3]
obs_cost = get_obs_cost(obstacles)
initial_traj = get_straight_line_traj(q_start, q_goal)

traj_progress, final_cost = run_chomp(
    initial_traj, obs_cost, 1, 1000, learning_rate=0.01, use_covariant_gradient=True
)
anim1 = visualize_traj_progress(traj_progress, obs_cost)
anim2 = play_traj(traj_progress[-1], obstacles)
```

Figure



Figure



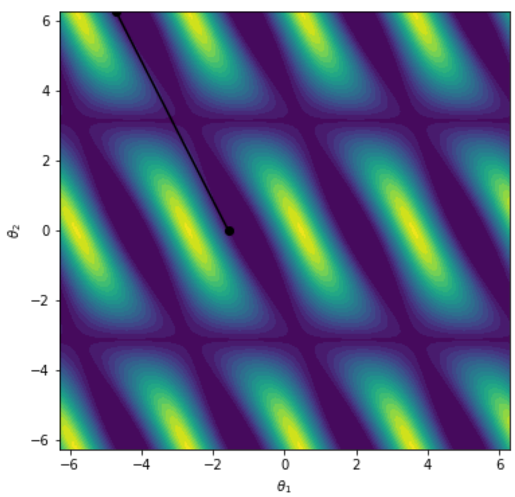
Implement the following function which automatically looks for the best goal point in C-space:

In [26]:
```
def run_goal_set_chomp(q_start, x_goal, y_goal, obs_cost, *args, **kwargs):
    """
    This function runs CHOMP for each C-space point corresponding to the goal point
    (x_goal, y_goal) in W-space. It returns the outputs of run_chomp for the C-space
    goal that results in the lowest final cost.
    """
    qs = inverse_kinematics(x_goal, y_goal)
    best_traj_process = None
    lowest_final_cost = np.inf
    for q_goal in qs:
      initial_traj = get_straight_line_traj(q_start, q_goal)
      traj_progress, final_cost = run_chomp(initial_traj, obs_cost, *args, **kwargs)
      if final_cost < lowest_final_cost:
        lowest_final_cost = final_cost
        best_traj_process = traj_progress
    return best_traj_process, lowest_final_cost
```

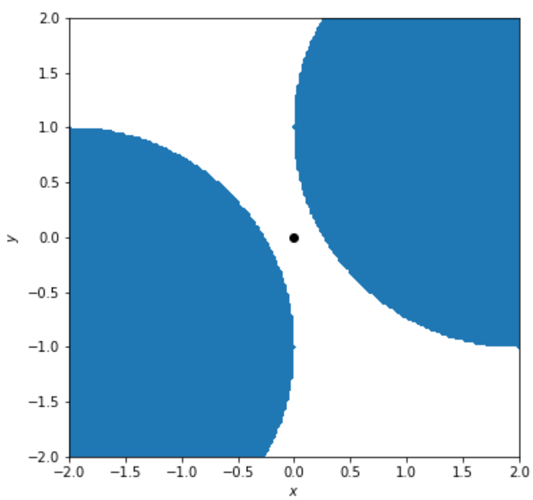
You can test your implementation with the following trajectory optimization problem:

```
In [27]: obstacles = circle(2, center=(2, 1)) | circle(2, center=(-2, -1))
         q_start = inverse_kinematics(0, -2)[0]
         obs_cost = get_obs_cost(obstacles)

         traj_progress, final_cost = run_goal_set_chomp(
             q_start, 0, 2, obs_cost, 1, 1000, learning_rate=0.01, use_covariant_gradient=True
         )
         anim1 = visualize_traj_progress(traj_progress, obs_cost)
         anim2 = play_traj(traj_progress[-1], obstacles)
```

Figure



Figure



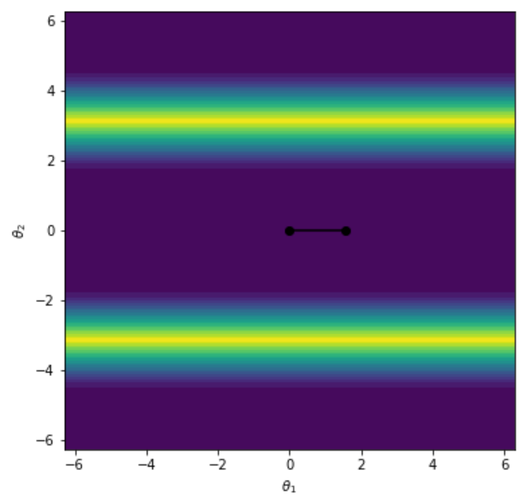
Adding a human-aware cost function

For the final part of the homework, we'll finally put the "human" in algorithmic human-robot interaction. Suppose that, like in the [Mainprice et al. paper (https://ieeexplore.ieee.org/document/5980048)](https://ieeexplore.ieee.org/document/5980048) that was presented in class, we want to optimize for visibility by a human.

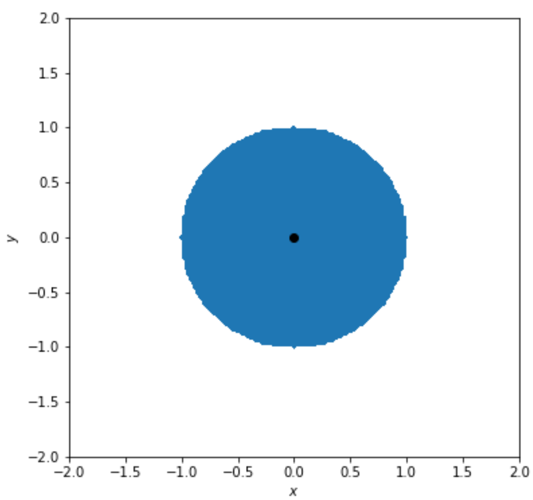
We'll start with this simple trajectory optimization problem:

In [28]:
```
obstacles = circle(1, center=(0, 0))
q_start = inverse_kinematics(2, 0)[0]
obs_cost = get_obs_cost(obstacles)

traj_progress, final_cost = run_goal_set_chomp(
    q_start, 0, 2, obs_cost, 0.5, 100, learning_rate=0.01, use_covariant_gradient=True
)
anim1 = visualize_traj_progress(traj_progress, obs_cost)
anim2 = play_traj(traj_progress[-1], obstacles)
```

Figure



Figure



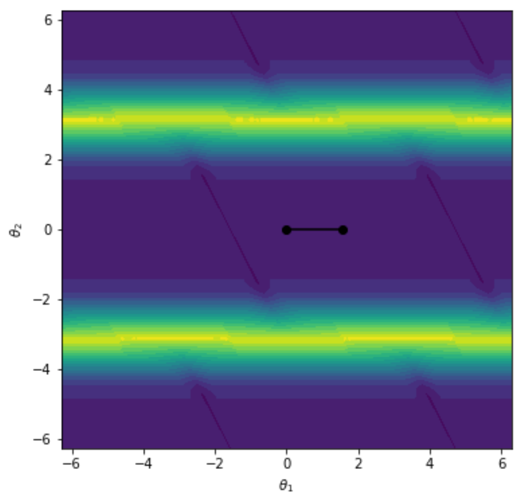
We can modify this problem by adding an additional cost term. In particular, let's say that there is a human standing at the point $(0, -2)$ and looking in the positive-y direction. We can use the following cost function to incentivize the robot to follow a trajectory that is more visible to the human:

$$c_{\text{visible}}(x, y) = \begin{cases} |x| - 1/2 & |x| \leq 1/2, y \leq 0 \\ 0 & \text{otherwise} \end{cases}$$

Modifying the below code to add $0.2 \times c_{\text{visible}}$ to the obstacle cost function before passing it to `run_goal_set_chomp` .

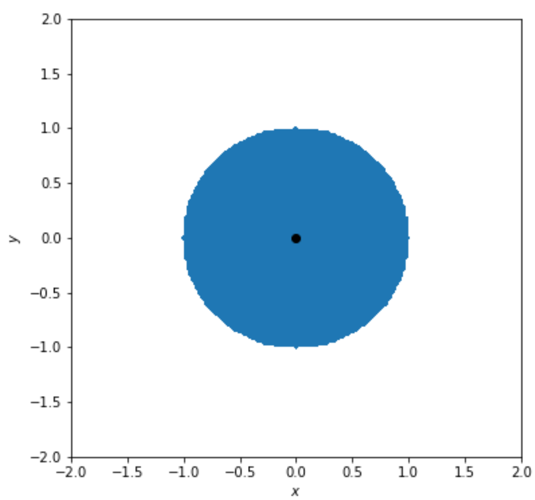
In [29]:
```
obstacles = circle(1, center=(0, 0))
q_start = inverse_kinematics(2, 0)[0]
obs_cost = get_obs_cost(obstacles)
def c_vis(theta1,theta2):
  x,y = forward_kinematics(theta1, theta2)
  if np.abs(x) <= 1/2 and y <= 0:
    return np.abs(x) - 1/2
  return 0

for i in range(obs_cost.shape[0]):
  for j in range(obs_cost.shape[0]):
    obs_cost[i][j] += 0.2 * c_vis(theta1s[i], theta2s[j])

traj_progress, final_cost = run_goal_set_chomp(
    q_start, 0, 2, obs_cost, 0.5, 100, learning_rate=0.01, use_covariant_gradient=True
)
anim1 = visualize_traj_progress(traj_progress, obs_cost)
anim2 = play_traj(traj_progress[-1], obstacles)
```

Figure




```
<ipython-input-11-e378fb46ebd4>:23: RuntimeWarning: More than 20 figures have been opened. Figures created through t
he pyplot interface (`matplotlib.pyplot.figure`) are retained until explicitly closed and may consume too much memor
y. (To control this warning, see the rcParam `figure.max_open_warning`).
  fig, ax = plt.subplots(figsize=(6, 6))
```

Figure



In [29]:

```

```

How is the resulting trajectory different from the result without using the c_{visible} term?

The resulting trajectory spent significantly more time close to where the human observer is (x=0, y=-2) compared to the trajectory without the c_{visible} *term*

In [29]:

```

```