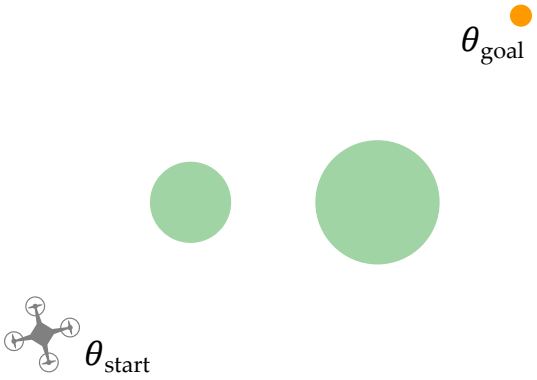
Problem Set 8

Robotics & Automation
Dylan Losey, Virginia Tech

Instructions. Please write legibly and do not attempt to fit your work into the smallest space possible. It is important to show all work, but basic arithmetic can be omitted. You are encouraged to use Matlab when possible to avoid hand calculations, but print and submit your commented code for non-trivial calculations. You can attach a pdf of your code to the homework, use live scripts or the publish feature in Matlab, or include a snapshot of your code. Do not submit .m files — we will not open or grade these files.

1 Potential Fields



In this problem you will use potential fields to get a motion plan for the 2-DoF environment shown above. Here the drone's position is $\theta = [x, y]^T$.

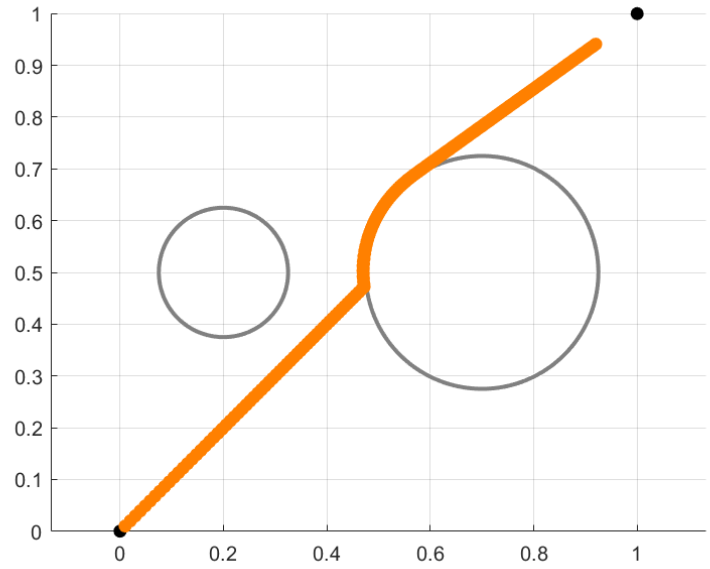
1.1 (15 points)

Implement the potential fields approach:

- Set $\theta_{start} = [0, 0]^T$ and $\theta_{goal} = [1, 1]^T$
- The first obstacle has center $c_1 = [0.2, 0.5]^T$ and radius $r_1 = 0.125$
- The second obstacle has center $c_2 = [0.7, 0.5]^T$ and radius $r_2 = 0.225$
- **Hint:** Start with a low learning rate α in your gradient descent algorithm. The result shown below was obtained with $\alpha = 0.01$.

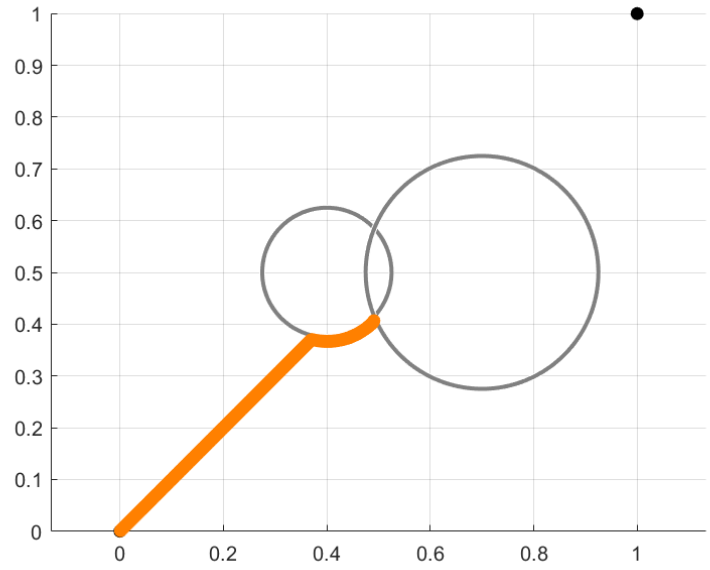
The motion plan must reach a final position within 0.1 units of the goal. Turn in your code and a plot of the result using **Publish** in Matlab. Visualize the obstacles in your plot: your solution should look like the example below.

> Your published code must include a motion plan similar to the one shown at the top of the next page. The correct code is listed at the end of Problem 1.



1.2 (10 points)

Modify the position of the obstacles so that a valid plan from θ_{start} to θ_{goal} exists but the potential fields planner fails (i.e., gets stuck). Turn in a **plot** that shows the obstacles and the failed motion plan. **Explain** why potential fields fail in your environment.



There are many possible solutions. For instance, set $c_1 = [0.4, 0.5]^T$. Here the motion plan gets stuck trying to go between the two obstacles: see the plot above. This failure occurs because the robot uses gradient descent to move towards decreasing potential energy, and the robot gets trapped in a **local minimum**.

```
clear
close all

% parameters
theta_start = [0; 0];
theta_goal = [1; 1];
centers = [.3, .7; .5, .5];
radii = [.125, .225];

% create figure
figure
grid on
hold on
for idx = 1:length(radii)
    viscircles(centers(:,idx)', radii(idx), 'Color', [0.5, 0.5, 0.5]);
end
plot(theta_start(1), theta_start(2), 'ko', 'MarkerFaceColor', 'k')
plot(theta_goal(1), theta_goal(2), 'ko', 'MarkerFaceColor', 'k')
axis equal

% gradient descent down potential field
theta = theta_start;
delta = 0.01;
learning_rate = 0.01;
for idx = 1:1000
    if norm(theta - theta_goal) < 0.1
        break
    end
    U = field(theta, theta_goal, centers, radii);
    U1 = field(theta + [delta; 0], theta_goal, centers, radii);
    U2 = field(theta + [0; delta], theta_goal, centers, radii);
    Ugrad = [U1 - U; U2 - U] / delta;
    theta = theta - learning_rate * Ugrad;
    plot(theta(1), theta(2), 'o', 'color', [1, 0.5, 0], ...
        'MarkerFaceColor', [1, 0.5, 0])
end

% find potential field at position theta
function U = field(theta, theta_goal, centers, radii)
    U = 0.5 * norm(theta_goal - theta)^2;
    for idx = 1:length(radii)
        center = centers(:, idx);
        radius = radii(idx);
        dist = norm(center - theta);
        if dist < radius
            U = U + 0.5 * (1/dist - 1/radius)^2;
        end
    end
end
```

2 Trajectory Optimization

In this problem you will use trajectory optimization to perform motion planning in 2-DoF environments. As before, the mobile robot's position is $\theta = [x, y]^T$.

2.1 (15 points)

Implement the trajectory optimization algorithm. Your code should be able to work with an arbitrary number of waypoints and circular obstacles. Set the initial trajectory ξ^0 as:

```
xi_0 = [linspace(theta_start(1), theta_goal(1), k);
        linspace(theta_start(2), theta_goal(2), k)];
```

> See the published code at the end of Problem 2. Notice that `centers` is a matrix where each column corresponds to an obstacle center. Similarly, `radii` is a vector where the i-th entry corresponds to the radius of the i-th obstacle.

2.2 (15 points)

Use your code to find a desired trajectory for the following environments. In each environment $\theta_{start} = [0, 0]^T$ and $\theta_{goal} = [1, 1]^T$.

- **Environment 1.** One obstacle with center $c_1 = [0.55, 0.5]^T$ and radius $r_1 = 0.3$. Your trajectory should have $k = 10$ waypoints.
- **Environment 2.** One obstacle with center $c_1 = [0.5, 0.3]^T$ and radius $r_1 = 0.3$. A second obstacle with center $c_2 = [0.5, 0.7]^T$ and radius $r_2 = 0.2$. Set $k = 15$.
- **Environment 3.** One obstacle with center $c_1 = [0.2, 0.35]^T$ and radius $r_1 = 0.2$. A second obstacle with center $c_2 = [0.5, 0.3]^T$ and radius $r_2 = 0.2$. A third obstacle with center $c_3 = [0.7, 0.5]^T$ and radius $r_3 = 0.2$. Set $k = 20$.

Turn in a **published** version of your code and plots. Each plot should visualize the obstacles, show the initial trajectory ξ^0, and show the optimal trajectory ξ. For instance, your solution for the first environment should look like the plot below.

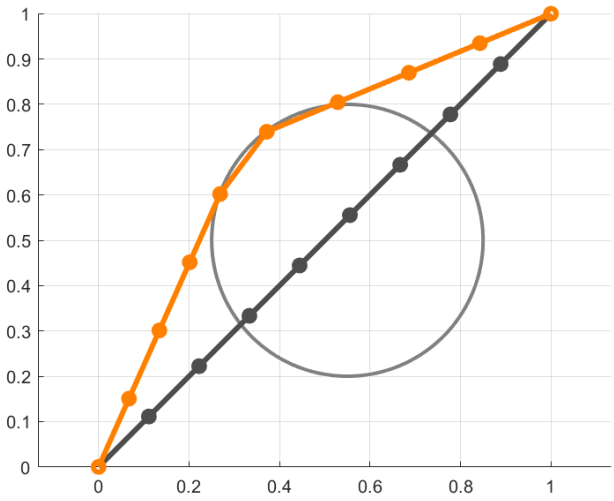


Figure 1: Trajectory optimization solution for **Environment 1**

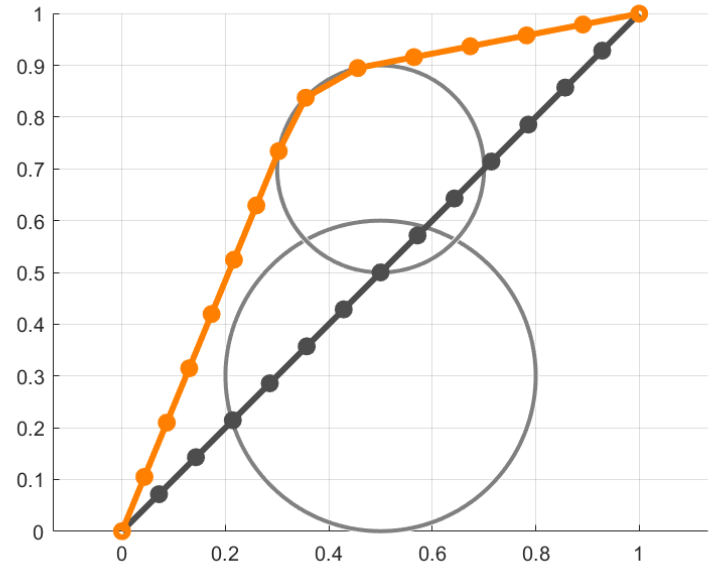


Figure 2: Trajectory optimization solution for **Environment 2**

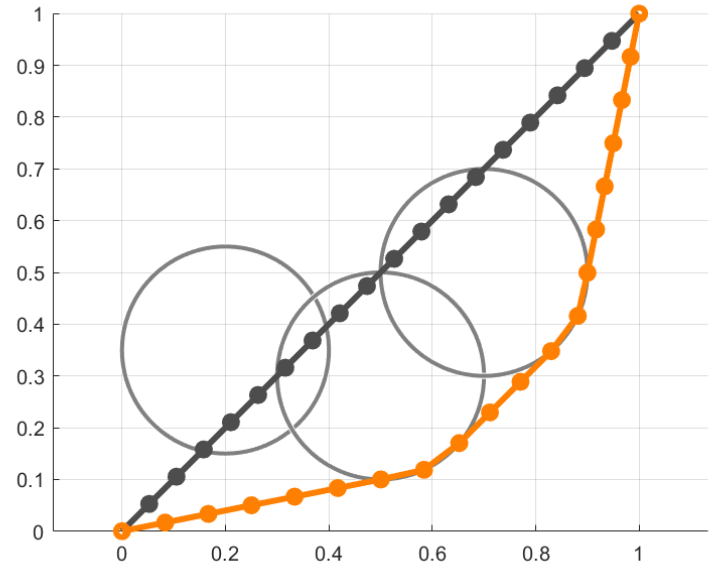


Figure 3: Trajectory optimization solution for **Environment 3**

See the trajectory plots in Figure 1, Figure 2, and Figure 3. Here the gray line is the initial trajectory ξ^0, while the orange line is the optimal trajectory ξ. The code used to produce these figures is listed at the end of Problem 2. If your solution goes (for example) above the obstacles instead of the below the obstacles, that is fine.

2.3 (10 points)

Consider an environment with two obstacles:

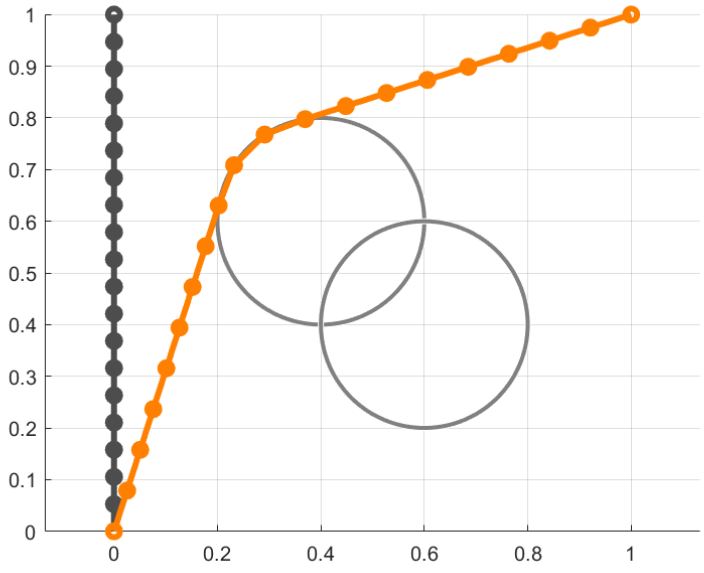
- $\theta_{start} = [0, 0]^T$ and $\theta_{goal} = [1, 1]^T$
- First obstacle with center $c_1 = [0.4, 0.6]^T$ and radius $r_1 = 0.2$
- Second obstacle with center $c_2 = [0.6, 0.4]^T$ and radius $r_2 = 0.2$
- The trajectory ξ should have $k = 20$ waypoints

Modify the initial trajectory ξ^0 so that the optimal trajectory goes around both obstacles. Submit a **plot** of your result and **list** the initial trajectory that you used.

For this environment an initial trajectory ξ^0 that moves straight from start to goal gets stuck. The nonlinear optimizer cannot find a way to modify this initial trajectory to decrease the cost (i.e., we are stuck in a local minima), and so the final answer simply *jumps* across the obstacles. We can fix this problem by choosing an initial trajectory that moves either above or below the obstacles. For example, try:

$$\xi^0 = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 1/(k-1) & 2/(k-1) & \dots & 1 \end{bmatrix}, \quad \xi^0 \in \mathbb{R}^{n \times k} \tag{1}$$

This results in an optimal trajectory ξ that goes above both obstacles. See the figure below. If you selected an initial trajectory that caused the robot to converge to a path below both obstacles, that is also fine.

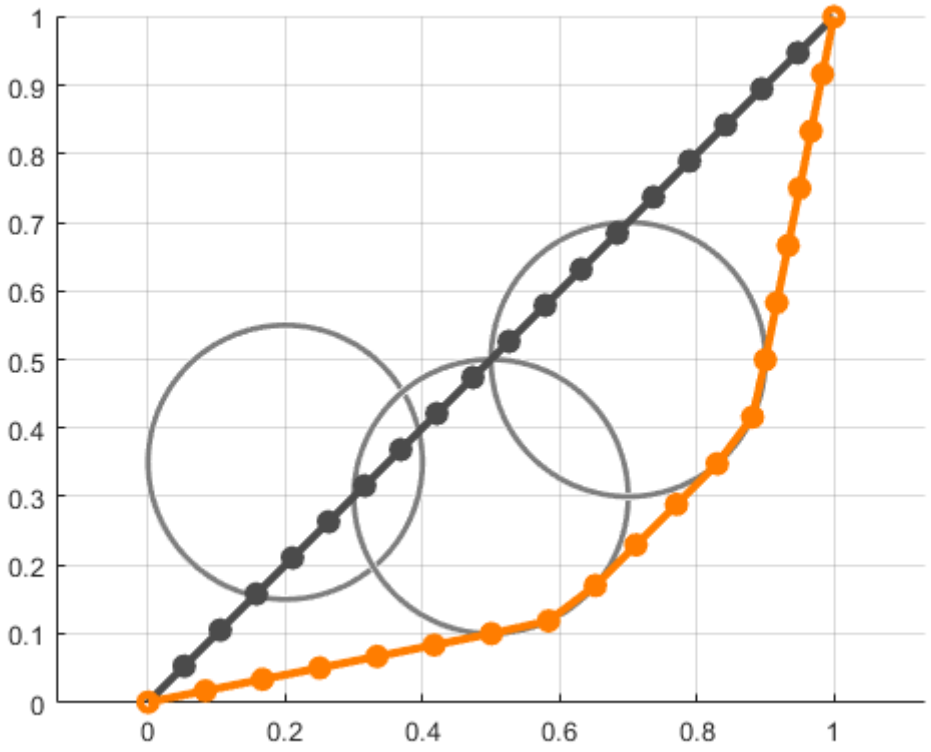


```
clear
close all

% start and goal
theta_start = [0;0];
theta_goal = [1;1];
centers = [0.2 0.5, 0.7; 0.35 0.3, 0.5];
radii = [0.2, 0.2, 0.2];

% initial trajectory
n = 2;
k = 20;
xi_0 = [linspace(theta_start(1), theta_goal(1), k);...
    linspace(theta_start(2), theta_goal(2), k)];
xi_0_vec = reshape(xi_0, [], 1);

% start and goal equality constraints
A = [eye(n) zeros(n, n*(k-1));...
    zeros(n, n*(k-1)), eye(n)];
B = [theta_start; theta_goal];

% nonlinear optimization
options = optimoptions('fmincon','Display','final',...
    'Algorithm','sqp','MaxFunctionEvaluations',1e5);
xi_star_vec = fmincon(@(xi) cost(xi, centers, radii), xi_0_vec, ...
    [], [], A, B, [], [], [], options);
xi_star = reshape(xi_star_vec, 2, []);

% plot result
figure
grid on
hold on
axis equal
for idx = 1:length(radii)
    viscircles(centers(:, idx)', radii(idx), 'Color', [0.5, 0.5,
 0.5]);
end
plot(xi_0(1,:), xi_0(2,:), 'o-', 'Color', [0.3, 0.3,
 0.3], 'LineWidth', 3);
plot(xi_star(1,:), xi_star(2,:), 'o-', 'Color', [1, 0.5,
 0], 'LineWidth', 3);

% cost function to minimize
function C = cost(xi, centers, radii)
    xi = reshape(xi, 2, []);
    C = 0;
    for idx = 2:length(xi)
        theta_curr = xi(:, idx);
        theta_prev = xi(:, idx - 1);
        C = C + norm(theta_curr - theta_prev)^2;
        for jdx = 1:length(radii)
            center = centers(:, jdx);
```

```
            radius = radii(jdx);
            if norm(theta_curr - center) < radius
               C = C + 0.5*(1/norm(theta_curr - center) - 1/radius)^2;
            end
        end
    end
end
```

Local minimum possible. Constraints satisfied.

fmincon stopped because the size of the current step is less than the value of the step size tolerance and constraints are satisfied to within the value of the constraint tolerance.



Published with MATLAB® R2020b

3 RRT Algorithm

In this problem you will use the RRT algorithm to perform motion planning in 2-DoF environments. As before, the mobile robot's position is $\theta = [x, y]^T$.

3.1 (10 points)

Implement the RRT algorithm. Your code should be able to work with an arbitrary number of circular obstacles.

- The bounds of the workspace are $x \in [0, 1]$, $y \in [0, 1]$
- The motion plan must end within $\epsilon = 0.1$ units of the goal.

> See the published code at the end of Problem 3. Notice that `centers` is a matrix where each column corresponds to an obstacle center. Similarly, `radii` is a vector where the i-th entry corresponds to the radius of the i-th obstacle.

3.2 (15 points)

Use your code to find a desired trajectory for the following environments. In each environment $\theta_{start} = [0, 0]^T$ and $\theta_{goal} = [1, 1]^T$.

- **Environment 1.** One obstacle with center $c_1 = [0.55, 0.5]^T$ and radius $r_1 = 0.3$.
- **Environment 2.** One obstacle with center $c_1 = [0.5, 0.3]^T$ and radius $r_1 = 0.3$. A second obstacle with center $c_2 = [0.5, 0.7]^T$ and radius $r_2 = 0.2$.
- **Environment 3.** One obstacle with center $c_1 = [0.2, 0.35]^T$ and radius $r_1 = 0.2$. A second obstacle with center $c_2 = [0.5, 0.3]^T$ and radius $r_2 = 0.2$. A third obstacle with center $c_3 = [0.7, 0.5]^T$ and radius $r_3 = 0.2$.

Turn in a **published** version of your code and plots. Each plot should visualize the obstacles, show the tree G, and show the final motion plan. For instance, your solution for the first environment should look like the plot below.

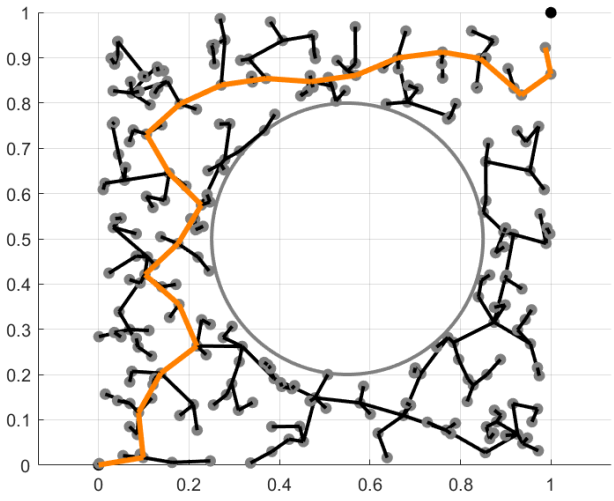


Figure 4: RRT solution for **Environment 1**

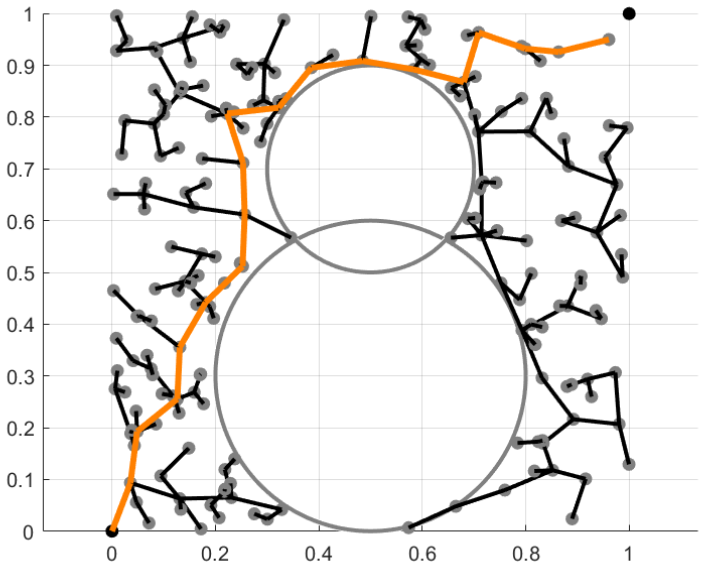


Figure 5: RRT solution for **Environment 2**

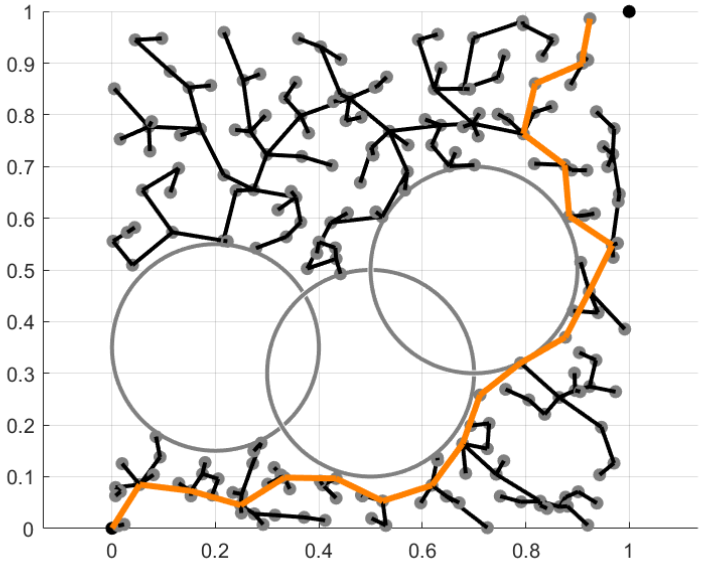


Figure 6: RRT solution for **Environment 3**

See the trajectory plots in Figure 4, Figure 5, and Figure 6. Here the black lines and gray dots show the tree G, while the orange line is the final motion plan from θ_{start} to a point close to the goal ($\epsilon \leq 0.1$). The code used to produce these figures is listed at the end of Problem 3. Each time you run your RRT code you should get a different solution: RRT builds the tree through random sampling.

3.3 (10 points)

Using **Environment 3** from the previous part, compare two versions of RRT. The first version is the standard RRT algorithm you have implemented (let's refer to this as the **baseline**). The second version will sample the goal more frequently (let's refer to this as **goal bias**). For **goal bias**, with probability 0.2 set θ_{rand} as θ_{goal}. Otherwise sample randomly as normal.

Run your code 10 times for **baseline** and 10 times for **goal bias**. Write down how many samples it takes on average to find a motion plan. Which approach is more sample-efficient: **baseline** or **goal bias**? Write a few sentences to explain and support your answer.

The samples N across ten runs with **baseline** and **goal bias** are tabulated below.

run	**baseline**	**goal bias**
1	254	225
2	363	169
3	318	150
4	590	143
5	352	249
6	359	166
7	202	251
8	267	162
9	440	183
10	226	313

On average, the **baseline** required 337 samples to reach a valid motion plan, while **goal bias** needed 201 samples. For **Environment 3** the evidence suggests that biasing the samples towards θ_{goal} decreases the total number of samples and causes RRT to reach a solution faster. Intuitively, this is because the tree is trying to move in the direction of the goal more frequently. This is particularly advantageous when the robot is in free space (or has passed the obstacles) and should move directly towards the goal. The code snippet required to implement **goal bias** is shown below.

```
% sample random joint position
if rand < 0.2
    theta_rand = theta_goal.coord;
else
    theta_rand = rand(2,1);
end
```

```
clear
close all

% environment
theta_start.coord = [0; 0];
theta_goal.coord = [1; 1];
centers = [0.2, 0.5, 0.7; 0.35, 0.3, 0.5];
radii = [0.2, 0.2, 0.2];

% parameters
epsilon = 0.1;
delta = 0.1;
N = 1000;

% visualize environment
figure
grid on
hold on
axis([0, 1, 0, 1])
axis equal
for idx = 1:length(radii)
    viscircles(centers(:, idx)', radii(idx), 'Color', [0.5, 0.5,
 0.5]);
end
plot(0, 0, 'ko', 'MarkerFaceColor', 'k')
plot(1, 1, 'ko', 'MarkerFaceColor', 'k')

% initialize tree
theta_start.parent = 0;
G(1) = theta_start;

for idx = 1:N

    % stop if theta_new is close to theta_goal
    if norm(G(end).coord - theta_goal.coord) < epsilon
        break
    end

    % sample random joint position
    theta_rand = rand(2,1);

    % find node in G nearest to theta_rand
    dist = zeros(length(G), 1);
    for jdx = 1:1:length(G)
        dist(jdx) = norm(G(jdx).coord - theta_rand);
    end
    [~, theta_near_index] = min(dist);
    theta_near = G(theta_near_index);

    % take a step from theta_near towards theta_rand
    vec_to_rand = theta_rand - theta_near.coord;
    dist_to_rand = norm(vec_to_rand);
```

```
    if dist_to_rand < delta
        theta_new.coord = theta_rand;
    else
        theta_new.coord = theta_near.coord + delta * ...
            vec_to_rand/dist_to_rand;
    end

    % check if theta_new is collision free
    collision = false;
    for jdx = 1:length(radii)
        center = centers(:, jdx);
        radius = radii(jdx);
        if norm(theta_new.coord - center) < radius
           collision = true;
        end
    end
    if collision
        continue
    end

    % if collision free, add theta_new to tree with parent theta_near
    theta_new.parent = theta_near_index;
    G = [G, theta_new];

    % plot node and edge
    plot(theta_new.coord(1), theta_new.coord(2), 'o', 'Color',  [0.5,
 0.5, 0.5], 'MarkerFaceColor', [0.5, 0.5, 0.5])
    line([theta_near.coord(1), theta_new.coord(1)],
 [theta_near.coord(2), theta_new.coord(2)], 'Color', 'k', 'LineWidth',
 2);

end

% work backwards from the final node to the root of the tree
next_theta = G(end);
while next_theta.parent ~= 0
    prev_theta_idx = next_theta.parent;
    prev_theta = G(prev_theta_idx);
    line([next_theta.coord(1), prev_theta.coord(1)],
 [next_theta.coord(2), prev_theta.coord(2)], 'Color', [1, 0.5,
 0], 'LineWidth', 3);
    next_theta = prev_theta;
end
```