ECE 470: Final Project - Swarm Navigation

Name: Ram Reddy
NetID: ramr3
Lab Partner: Kate Lee
TA: Lukas Zscherpel
Section: Monday 9AM
Date Submitted:12/15/2025

1. Introduction

The purpose of this lab is to engineer a multi-robot navigation function by thinking, implementing and testing a solution. The navigation function U will help navigate many robots to their goals by accounting for other robots and obstacles. This gradient function will be implemented in Python within the navigation function of the project. By accounting for all the separate possible collisions within the simulation like robots and other obstacles, each robot will get to the goal position. After successfully implementing the function, the navigation function should be tested across multiple separate configuration and target permutations.

The simulation itself is represented as a unit disk and each robot starts with 5 separate start positions on the unit circle ($\theta k = k\pi/n$) and will each go to separate goal destinations ($\theta' k = \sigma k\pi/n + \pi$) where σ is a permutation of {1, …, n!} and in this n is 5 so 120 target permutations per configuration. Each robot is represented as a disk of radius r and each obstacle is represented by radius R. There are a total of 120 starting permutations for the robots and can start in any of the start positions. The simulation will call the navigation function in every step and this will decide the direction each robot is moving towards by returning new velocities for each robot. If a robot touches or collides with another robot this means that the distance between the two robots ($\| x_i - x_j \|$) is within <= 2r where r is radius of the robots. x represents the center of the robot in the previous example. If the robot touches or collides with another obstacle this means the distance between the two ($\| x_k - c_m \|$) is within <= R_m + r where R is the radius of the obstacle and r is the radius of the robot. c represents the center of the obstacle in the previous example. r is defined to be greater than 0. Another collision occurs if any of the robots distance from the center of the unit circle is > 1.

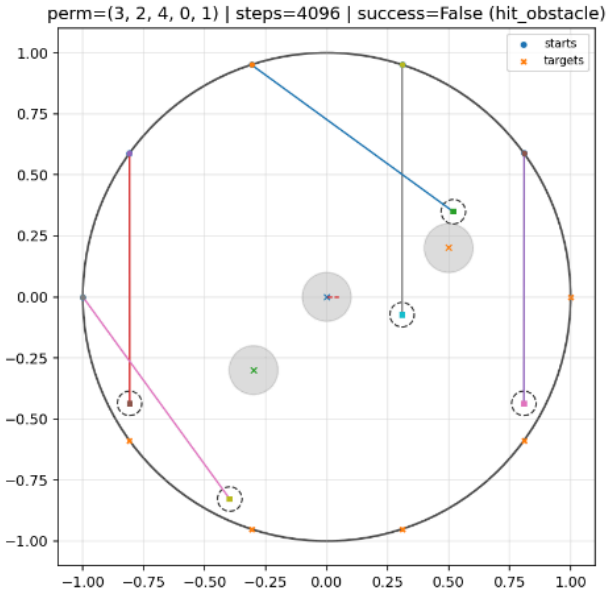

Figure 1: An Example of a Collision and the Workspace of the Simulation for C0 Configuration

Our primary goals for this project were to minimize failures that arise of collisions and stalling caused by minima with gradient functions, and reducing steps so that the simulation finishes within the 30k step limit.

When designing the navigation function we went step by step in solving each component of the big problem. We started with having a basic gradient function that would take the robot straight to the target and then slowly add collision detection so that the robots would move smoothly past them. With this approach we could make sure the navigation was robust in that it handled all possible cases and also solved simulation failures like stalling or collisions. In the end we tuned the barriers of the obstacles and robots to decrease steps and increase speed.

2. Methods

2.1 The Navigation Function U

The overall navigation function is defined by

$$U(X) = U_{goal}(X) + \lambda_{obs}U_{obs}(X) + \lambda_{pair}U_{pair}(X) + U_{boundary}(X)$$

where boundary is optional. The potential function is essentially a sum of attraction and repulsive
potential fields. In the code the velocity vector is essentially calculated from $V \approx \nabla U$.

First let us define important terms. Target attraction ($U_{goal}(X)$) is the potential component which pulls the robot to the goal and it should decrease monotonically as the robot gets closer to the

goal. This is meant to be simple and should make the robot directly move to the target as a straight line. The gradient itself is a straight line that is linearly scaled with the distance the robot is from the goal. Some tradeoffs of this method is that if the line goes straight through a narrow passage then the robot might get stuck without a place to go or might oscillate in between because of the repulsion forces of the other components in the equation.

The obstacle barrier ($\lambda_{obs}U_{obs}(X)$) is a repulsive potential which pushes robots away from the obstacles, the repulsion increases exponentially as it gets closer to the obstacle. This exponential repulsive potential causes the robot to slow down dramatically and head the opposite direction. Due to the very extreme nature of the exponential repulsive potential, the robot can possibly get stuck due to an oscillation between going away from the obstacle and moving towards the goal which makes the robot stuck in a local minima.

Pairwise separation ($\lambda_{pair}U_{pair}(X)$) is the robot to robot collision avoidance potential to make sure robots do not collide with each other. This repulsion potential also increases exponentially as two robots get closer together. This is very similar to the obstacle barrier as this exponential repulsive potential causes the robot to slow down dramatically and head the opposite direction. This also has the same issue which makes the robot possibly get stuck in a local minima.

Boundary handling is a potential that makes sure robots are kept within the unit circle workspace. This is not specifically implemented in our swarm navigation system.

Stabilizers are implemented into our system. Stabilizers make sure that the repulsive potential does not become infinity when the robot is right at the boundary of the obstacle or another robot. This will cap the maximum magnitude by limiting the lowest possible buffer zone (distance between the robot to an obstacle or another robot). This limits the minimum distance between obstacles and robots, so this means that in extreme cases it is possible for robots to be within 2r distance and robots and obstacles to be within r+R distance.

Deadlock avoidance is implemented in that when either an obstacle or robot collision is about to occur, robots will move tangentially together to avoid a collision. This causes robots to slide past other robots and obstacles instead of pushing against them. This is a heuristic which solves local minima, but might cause non optimal paths to targets and might also cause robot to robot loops.

2.2 The Computation for the Navigation Function

For the target attraction ($U_{goal}(X)$) where $V_{target,x}$= $target_x$ - $robot_x$ for the x direction and is represented in code below:

```
for i, robot in enumerate(state):
     Vx, Vy = targets[i] - robot
```

Code Snippet 1

where the state contains the current x,y positions of the n robots.

The obstacle repulsion ($\lambda_{obs}U_{obs}(X)$) is calculated for each obstacle k and the repulsive potential contribution for each obstacle in the x direction is represented by equation 1 shown below.

$$\Delta V_{obs,x} = \left(\frac{1}{dist_k * buffer_k^3}\right) * (robot_x - obstacle_{k,x}) \quad (1)$$

where

$$dist_k = \sqrt{(robot_x - obstacle_{k,x})^2 + (robot_y - obstacle_{k,y})^2}$$

$$buffer_k = max(dist_k - R_{obs,k} - r_{robot}, \epsilon)$$

$\epsilon = 10^{-5}$ which is a stabilizer to prevent division by zero.

In code this is represented as:

```
for obstacle in obstacles:
      xDiff = robot[0] - obstacle["center"][0]
      yDiff = robot[1] - obstacle["center"][1]

      dist = math.hypot(xDiff, yDiff)
      buffer = dist - (obstacle["radius"] + r)

      if buffer > obstacleBuffer:
            continue

      obstacleCollision = True
      buffer = max(buffer, 1e-5)
      mag = 1/(dist * pow(buffer, 3))
      Vx += mag * xDiff
      Vy += mag * yDiff
```

Code Snippet 2

One extra thing to note in the code is the use of an "obstacleBuffer" that essentially adds a buffer zone around the obstacle, so that function only adds potentials when the robot is within a "obstacleBuffer" distance of the obstacle. The "obstacleBuffer" is set to 0.01. As shown below the obstacle barrier exponentially increases the potential as the robot gets closer to the obstacle due to the cubic in the denominator.

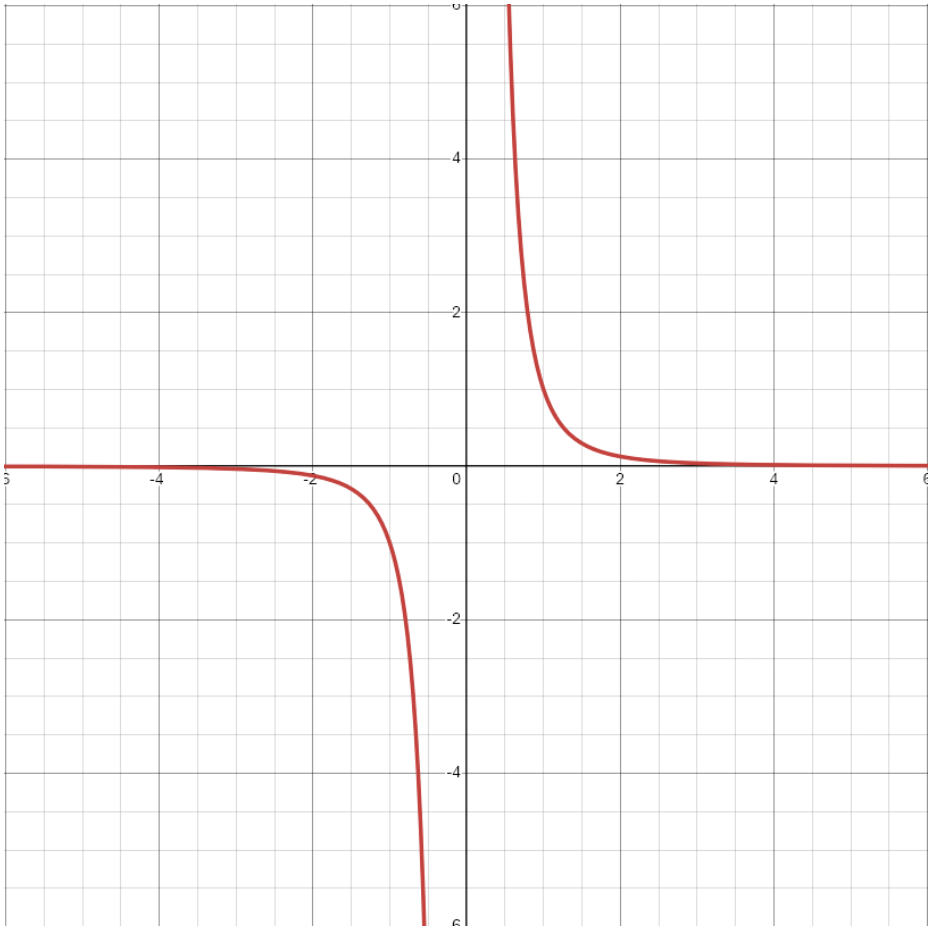

Figure 2: Visual Representation of Repulsion Function

This causes the robots to stay away from the obstacles.

The pairwise separation ($\lambda_{pair}U_{pair}(X)$), is calculated very similarly to the obstacle repulsion. Each pairwise gradient is calculated for each robot j and the repulsive potential contribution for each obstacle in the x direction is represented by equation 2 shown below.

$$\Delta V_{sep,x} = \left(\frac{1}{dist_j * buffer_j^3}\right) * (robot_x - robot_{j,x}) \quad (2)$$

where

$$dist_j = \sqrt{(robot_x - robot_{j,x})^2 + (robot_y - robot_{j,y})^2}$$

$$buffer_j = max(dist_j - r_{robot} - r_{robot}, \epsilon)$$

$\epsilon = 10^{-5}$ which is a stabilizer to prevent division by zero.

In code this is represented as:

```
for j in range(len(state)):
      ...

      robot2 = state[j]
      xDiff = robot[0] - robot2[0]
      yDiff = robot[1] - robot2[1]

      dist = math.hypot(xDiff, yDiff)
      buffer = dist - (2*r)

      if buffer > robotBuffer:
            continue

      robotCollision = True
      buffer = max(buffer, 1e-5)
      mag = 1/(dist * pow(buffer, 3))
      Vx += mag * xDiff
      Vy += mag * yDiff
```

Code Snippet 3

This similarly has a buffer called “robotBuffer” that essentially adds a buffer zone around the robot, so that function only adds potentials when the robot is within a “robotBuffer” distance of the robot. The “robotBuffer” is set to 0.02. This uses a similar equation as above and as shown the robot barrier exponentially increases the potential as the robot gets closer to other robots due to the cubic in the denominator. Both buffer values were chosen as they proved to work best in removing minima and stalling.

Whenever the robot was within the buffer distance of either the robot or obstacle that specific iteration is marked as a collision. In order to avoid stalling of robots in local minima a simple heuristic was used which is shown in code below.

```
if robotCollision or obstacleCollision:
      check = False
      targetDir = targets[i] - robot
      if obstacleCollision:
            for obstacle in obstacles:
                  obsDir = np.array(obstacle["center"]) - robot
                if np.dot(targetDir, obsDir) > 0.05:
                        check = True
                        break

      if not check and robotCollision:
            for j in range(len(state)):
                  if i == j:
                        continue
                  robot2Dir = state[j] - robot
                  if np.dot(targetDir, robot2Dir) > 0.05:
                        check = True
                        break

       if check:
             Vx, Vy = -Vy, Vx  # tangent velocity
       else:
            norm = math.hypot(Vx, Vy)
            if norm > 1e-6:
                  x, Vy = Vxi + Vx/norm, Vyi + Vy/norm
            else:
                  Vx, Vy = Vxi, Vyi
```

Code Snippet 4

Whenever either the velocity in the X direction is far greater than the Y direction this would imply that the potential function completely overrides any Y velocity causing the robot to stall. In order to fix this, the Vx and Vy component velocities are switched in order to make a tangential vector. However, to decrease steps the robot automatically does not apply tangential velocity if there are no obstacles or robots in the path to the target; this is checked by doing the dot product on both the obstacle/robot 2 direction with the target direction vector. If the direction vectors are perpendicular then the dot product is zero. To allow for some error we checked if the dot product was > 0.05.

Finally, once all the velocities are calculated for each robot the velocities have to be normalized as the exponential function will outscale other velocities. Each velocity is normalized with equation 3 shown below.

$$V_{final} = \frac{V}{||V|| + \varepsilon} \quad (3)$$

Where $\epsilon = 10^{-12}$ so that there is no divide by zero error.

This caps the velocities to 1, so that the simulation doesn't have extremely high or low velocities.

To sum everything together, we plotted the gradient of our navigation function with the C0 configuration space as shown below.

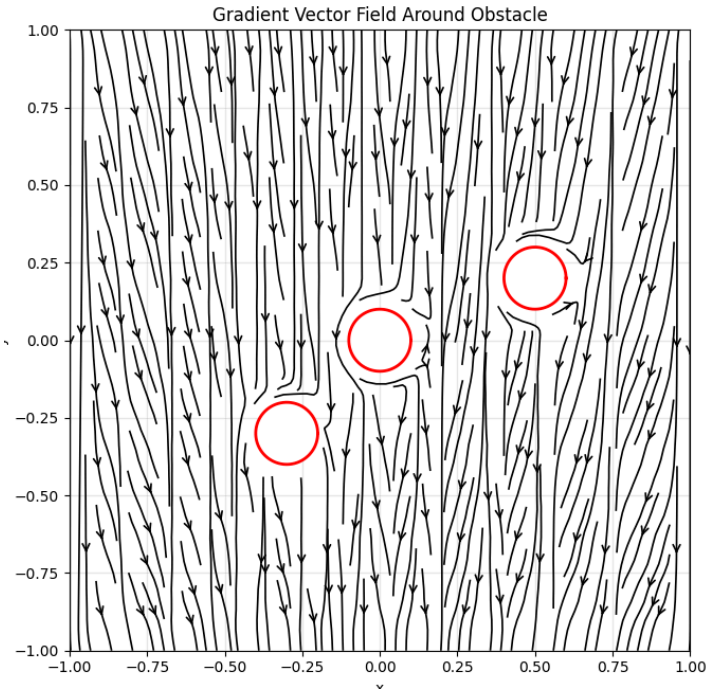

Figure 3: Combined Navigation Function Gradient Vector Field

3. Results

3.1 Defining Each Test

We tested a total of 5 test cases with 5 different obstacle configurations. The first obstacle configuration was the default C0 obstacle test case shown below

```
OBSTACLES = [
    {"center": (0.0, 0.0), "radius": 0.1},
    {"center": (0.5, 0.2), "radius": 0.1},
    {"center": (-0.3, -0.3), "radius": 0.1},
]
```

Code Snippet 5

and the workspace looking like the following:

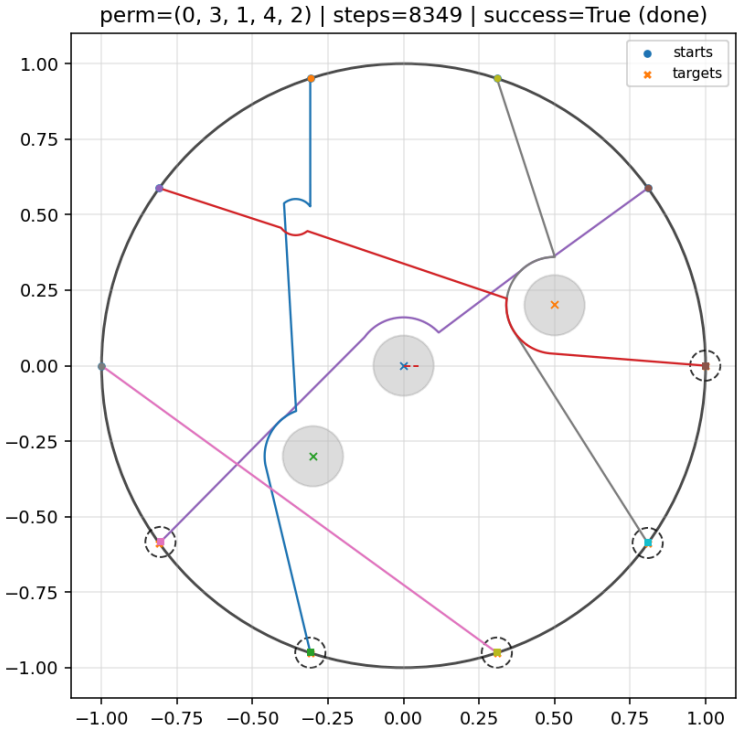

Figure 4: Default C0 Test (0, 3, 1, 4, 2) Trajectory

This test is useful as the obstacles are scattered around the workspace, so this will test a variety of useful things. It can test if the robots can avoid obstacles and robot to robot interactions. This will not necessarily stress test, but will ensure basic functionality of the robot controller..

The second test was designed to stress test the controller's ability to create paths when there are many U shaped local minima which are caused by 5 objects creating spaces where the robots can fit in. The configuration is shown below:

```
OBSTACLES = [
    {"center": (0.0, 0.0), "radius": 0.1},
    {"center": (0.15, 0.15), "radius": 0.08},
    {"center": (-0.15, 0.15), "radius": 0.1},
    {"center": (0.15, -0.15), "radius": 0.07},
    {"center": (-0.15, -0.15), "radius": 0.09},
]
```

Code Snippet 6

and the workspace looking like the following:

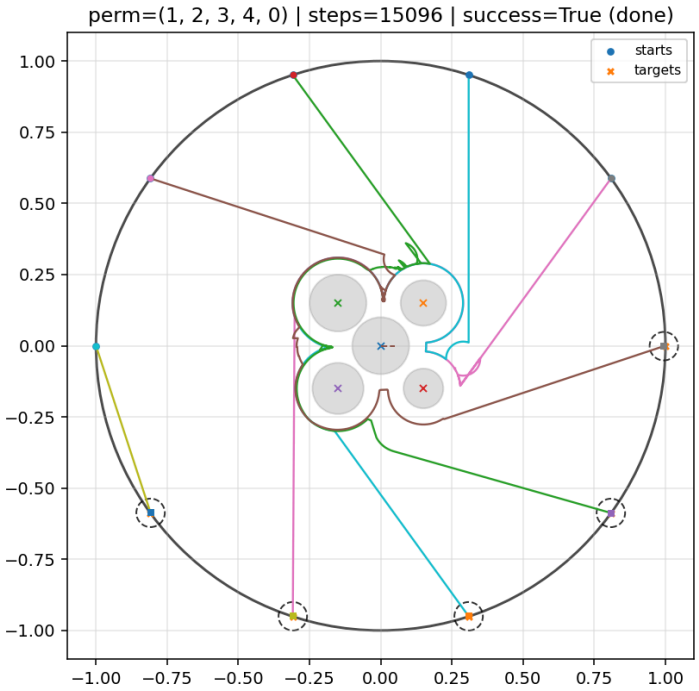

Figure 5: Stress U Shaped Test Permutation (1, 2, 3, 4, 0) Trajectory

The third test was to test if robots will get stuck if two obstacles create a tight crevice where robots cannot go through and the target is the other side. This makes sure that the controller can create paths without deadlock and stalling for each robot. It is shown with the following configuration:

```
OBSTACLES = [
    {"center": (0.5, 0.22), "radius": 0.2},
    {"center": (0.5, -0.22), "radius": 0.2},
]
```

Code Snippet 7

and the workspace looking like the following:

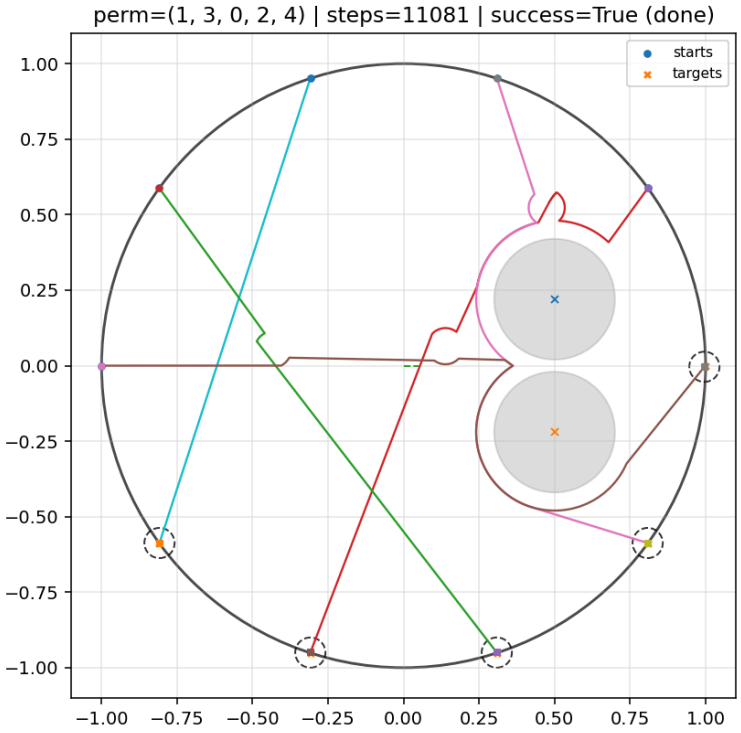

Figure 6: Test 3 Permutation (1, 3, 0, 2, 4) Trajectory

The fourth test was designed to stress test robot to robot interaction. In this test there is a single obstacle in the middle where robots must be able to interact with other robots and the obstacle correctly without collisions. The test is shown below:

```
OBSTACLES = [
    {"center": (0.0, 0.0), "radius": 0.05},
]
```

Code Snippet 8

and the workspace looking like the following:

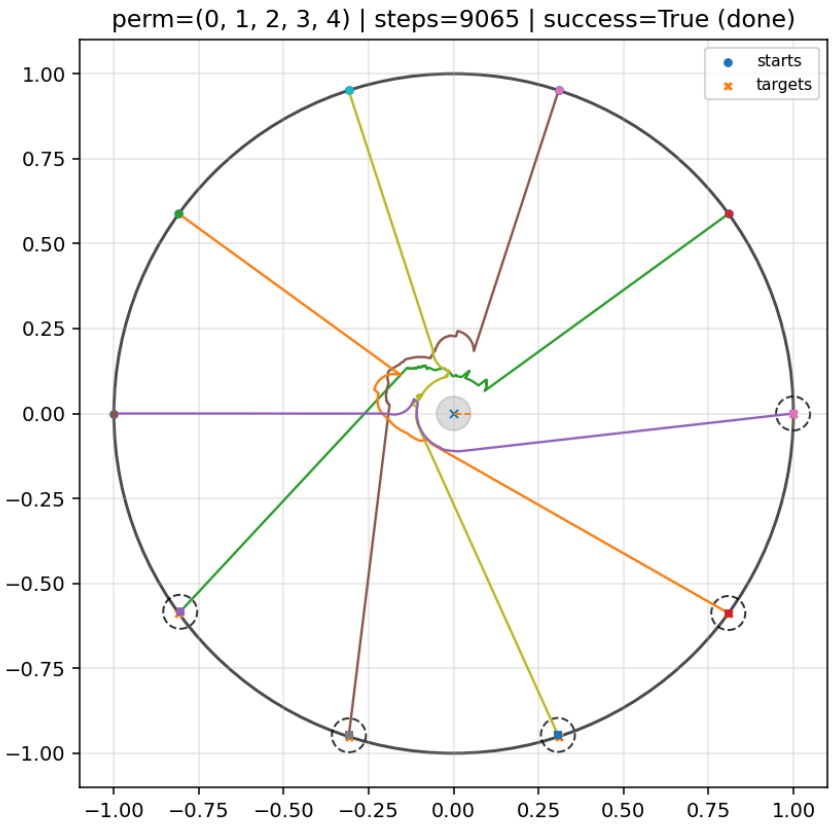

Figure 7: Robot to Robot Test Permutation (0, 1, 2, 3, 4) Trajectory

The fifth test was designed to create symmetric local minima and was also designed to test our deadlock detection. Due to the large size of the outside obstacles and small inside obstacles, robots can get stuck at the center if the repulsion potential is equal. Robots can also get stuck with the outside obstacles because of possible repulsion of other robots and other obstacles.

The configuration is shown below:

```
OBSTACLES = [
    {"center": (-0.35, 0.35), "radius": 0.2},
    {"center": (0.35, -0.35), "radius": 0.2},
    {"center": (0.35, 0.35), "radius": 0.2},
    {"center": (-0.35, -0.35), "radius": 0.2},
    {"center": (0.0, 0.0), "radius": 0.05},
]
```

Code Snippet 9

and the workspace looking like the following:

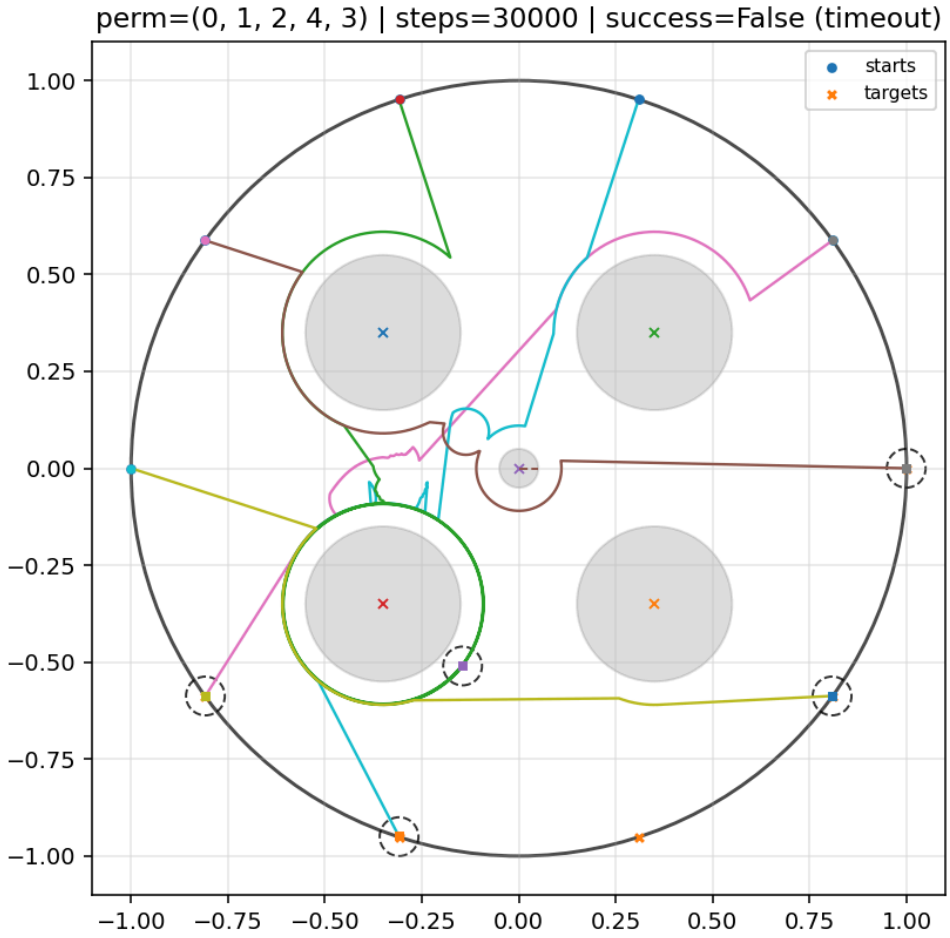

Figure 8: Symmetric Local Minima Test Failed Permutation (0, 1, 2, 4, 3) Trajectory

The number of robots (n) was kept constant at 5, the radius of each (r) was kept at 0.05, the time interval (dt) was kept at 0.005, the vmax was kept at 0.05, the tolerance (tol) was at 0.005 and the max number of steps (max_steps) was kept at 300000. This was done to keep each simulation consistent and meet with the specifications of the 5 test configurations given while grading.

3.2 Summary of Results

After conducting all the tests with the various obstacle configurations the data was compiled into the table below.

Table 1. Success Rate, Average Steps, Max and Min Steps for each Test				
Test Type	**Success Rate (%)**	**Average Steps**	**Max Steps**	**Min Steps**
Default C0 (Test 1)	100% (120/120)	9,794.98 ± 5,047	26635	6449
Stress U Shaped Minima (Test 2)	100% (120/120)	11,672.46 ± 4,557	24677	6449
Small Crevice (Test 3)	100% (120/120)	10,234.85 ± 2,353	15860	6449
Robot-Robot Interaction (Test 4)	100% (120/120)	8,450.42 ± 2,114	14906	6449
Stress Test Minima (Test 5)	94.17% (113/120)	13,355.84 ± 5,806	30000	6775

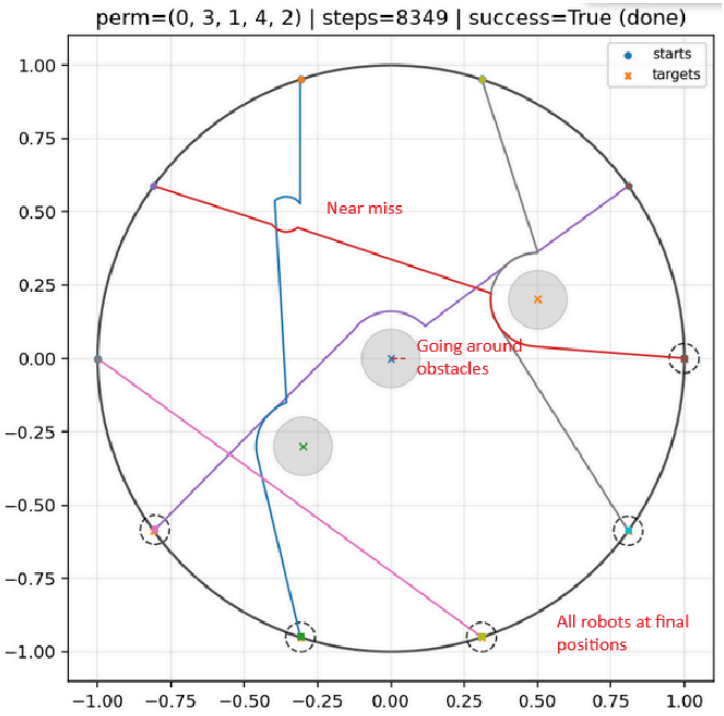

Figure 9: Annotated Default C0 Test (0, 3, 1, 4, 2) Trajectory

Figure 9 shows a typical run of the navigation function on the C0 test. It specifically shows the (0, 3, 1, 4, 2) permutation. We can see some of the default behavior with this test as at the top there is near miss for the blue and red robots. The navigation function automatically creates a tangential path so that each robot can go around each other. Another default behavior is shown as robots move around obstacles due to the high repulsion. This makes paths to be short and robots continuously moving no matter what.

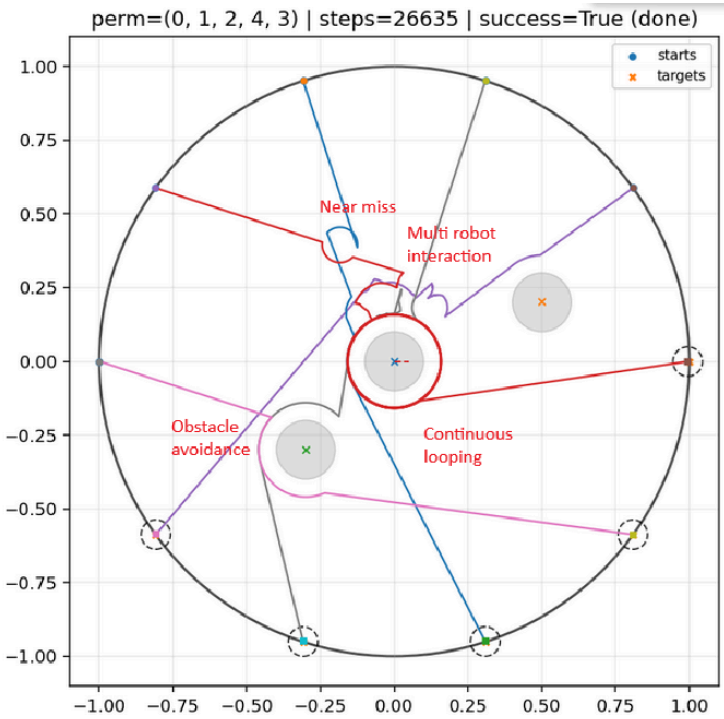

Figure 10: Annotated Default C0 Test (0, 1, 2, 4, 3) Trajectory

Figure 10 shows the (0, 1, 2, 4, 3) permutation of the C0 configuration. This is a more complex scenario and almost timed out with 26,635 steps. This similarly can deal with near misses shown with the blue and red trajectories. The example also shows multi-robot interaction with the very messy red, gray, and purple paths. Similarly to the previous, the example also shows obstacle avoidance with the pink path. The reason this specific permutation took so long to finish is because of the red path robot. We can see in the figure that the red path completely overrides all other paths implying that it had at least one circle around the obstacle. This is one of the pitfalls of using constant tangential velocity whenever there is a possibility of the target and obstacle direction being in the same direction. Ideally, the direction of the obstacle and target should only be checked for a single obstacle instead of all like in code snippet 4, however, this causes stalling as I will mention more in the discussion.

Overall, the navigation function works well in simple environments, getting to the optimal state fairly quickly. In more complex scenarios where there are more collisions and many local

minima, due to the tangential velocity algorithm it causes issues with constant looping around obstacles. Also, boundary repulsion was not added as it was optional, but it is possible if there were a series of obstacles leading to the barrier, the robot would just collide with the boundary. This method is fairly robust and will get to the target 100% of the time in most cases as shown in Table 1 with ~10k step average in most cases.

4. Discussion

The navigation function was successful in fundamental navigation tasks because of what we decided to use for the components of the function. In total 593/600 test cases passed with ~1% failure rate. Due to the exponential nature of Eq. 1 and Eq. 2 collisions are guaranteed to be avoided whether it is robot to robot or robot to obstacle. This extreme repulsion guarantees that the two cannot touch each other within a time step. There is also an extra buffer zone, so it has to either cross the obstacle or robot buffer to even collide. There is also stable velocity because all the velocities in the end become normalized to 1 due to Eq. 3. This ensures that the repulsion equation 1 and 2 cannot cause extreme jerky motions and instead will smooth out curves. The algorithm also solves stalling as there is tangential velocity heuristic added whenever velocities cancel out. This solves possible deadlocking by making sure robots evade each other rather than stop. Finally, whenever there is a lack of repulsion forces, the robot goes straight to the target as it's a linear function.

Initially there were a lot of collisions like the one shown in Figure 1, however due to Eq. 1 and Eq. 2 there were no more collisions between obstacles or robots.

The next problem we faced was stalling due to local minima, this was solved with the tangential heuristic as mentioned previously and can be seen used in Figure 10 as robots will go around obstacles rather than stick to one spot. This heuristic also solved the problem with oscillations and instability as the robot constantly moves to another location, however, this constant movement brings another problem of infinite looping around obstacles. This infinite looping can be seen in Figure 8 as the green path robot was looping around the bottom left obstacle. Eventually given enough steps the "infinite" loop stops as all other robots are in their goal destination. The best method to solve this problem is to change the heuristic, so that it only checks the nearest obstacle, but this introduces the old stalling problem again. Instead we used a bandaid solution that checks if there are no other obstacles or robots in the path to destination, this doesn't solve the problem fully, but does work the majority of the time.

Our algorithm goes more for speed rather than being conservative causing path length to be low. This in turn reduces safety, but due to the exponential nature of the function used it reduces possible collisions significantly. The algorithm is also more responsive, but adding the last normalization equation causes the function to be more stable.

If we had more time, then we would definitely like to solve the infinite loop problem fully and incorporate barrier repulsion to account for possible edge cases.

5. Conclusion

In the end, we were successfully able to implement a navigation function that has multi robot and obstacle avoidance. Our main goals were to minimize failures that arise of collisions and stalling caused by minima with gradient functions, and reducing steps so that the simulation finishes within the 30k step limit. From the combined 600 test cases run with complex minima and interactions only ~1% failed, so we minimized failures. Out of all the failures, the issue was infinite looping, so we also removed most if not all collisions and stalling with other obstacles and robots. We could not finish all tests within the 30k limit, so the main goal in the future is to remove infinite looping issues. I learned a lot in the project like how simulations work and how multi-entity interactions are solved. I also learned how to optimize and critically think of solutions to difficult issues like local minima and stalling.

Additional Resources

Final Project Report
Modern Robotics Textbook

Appendix A

```
import numpy as np
import math

def compute_gradients(state, targets, obstacles, r):
    """
     Compute a velocity field (interpreted by the envelope as -∇U) for each
robot.
    Inputs:
      state:    (n, 2) ndarray. Current xy positions for n robots.
      targets:  (n, 2) ndarray. Target xy positions for the same robots.
      obstacles: list of dicts: [{"center": (x, y), "radius": R}, ...]
        r:           float. REQUIRED minimum pairwise distance between any two
robots.
    Returns:
       grads:     (n, 2) ndarray. Velocity vectors for each robot at the current
state.
    """
    obstacleBuffer = 0.01
```

```
    robotBuffer = 0.02

    V = []
    for i, robot in enumerate(state):
        Vxi, Vyi = targets[i] - robot
        Vx, Vy = Vxi, Vyi  # distance between destinationand current location
        obstacleCollision = False
        robotCollision = False

        # obstacle repulsion
        for obstacle in obstacles:
            # difference between robot and obstacle
            xDiff = robot[0] - obstacle["center"][0]
            yDiff = robot[1] - obstacle["center"][1]

            # madnitude of that difference
            dist = math.hypot(xDiff, yDiff)
                # distance - radius of obstacle and robot
            buffer = dist - (obstacle["radius"] + r)

            # if the distance is still above some buffer value skip
            # 0.01 works after some testing
             if buffer > obstacleBuffer:  # no need to calculate for something
far away
                continue

            obstacleCollision = True

            buffer = max(buffer, 1e-5) # makes sure we dont divide by 0
            # uses 1/(distance * buffer^3)
            mag = 1/(dist * pow(buffer, 3))
             Vx += mag * xDiff # inital diff will be negative if robot on left
opposite if right
            Vy += mag * yDiff

        # robot to robot repulsion
        for j in range(len(state)):
            # skip same index robot
            if i == j:
                continue

            robot2 = state[j]
```

```
            # difference between robot and robot2
            xDiff = robot[0] - robot2[0]
            yDiff = robot[1] - robot2[1]

            # actual distance
            dist = math.hypot(xDiff, yDiff)
                # distance - radius of the two robots
            buffer = dist - (2*r)

             if buffer > robotBuffer:  # no need to calculate for something far
away
                continue

            robotCollision = True # mark current iteration as a collision

            buffer = max(buffer, 1e-5) # makes sure we dont divide by 0
            # uses 1/(distance * buffer^3)
            mag = 1/(dist * pow(buffer, 3))
            Vx += mag * xDiff
            Vy += mag * yDiff

        # if there might be a collision make sure robot does not get stuck
        if robotCollision or obstacleCollision:
            # check if obstacle or robot is block path to target
            check = False
            targetDir = targets[i] - robot

            if obstacleCollision:
                for obstacle in obstacles:
                      obsDir = np.array(obstacle["center"]) - robot  # direction
to obstacle
                      # check if obstacle is in same general direction as target
by using dot product
                    if np.dot(targetDir, obsDir) > 0.05:
                        check = True
                        break

            if not check and robotCollision:
                for j in range(len(state)):
                    if i == j:
                        continue
                    robot2Dir = state[j] - robot  # direction to other robot
```

```
                    # check robot dot product
                    if np.dot(targetDir, robot2Dir) > 0.05:
                        check = True
                        break

            if check:
                Vx, Vy = -Vy, Vx  # tangent velocity
            else:
                # normalize the repulsion and add to the Vx and Vy
                norm = math.hypot(Vx, Vy)
                if norm > 1e-6:
                    Vx, Vy = Vxi + Vx/norm, Vyi + Vy/norm
                else:
                    Vx, Vy = Vxi, Vyi

        V.append([Vx, Vy])

    norms = np.linalg.norm(V, axis=1, keepdims=True) + 1e-12 # add variability
to velocities by adding some constant
    return V / norms  # unit direction toward target
```