# POTENTIAL FIELD PLANNING

MEAM 520 LAB-04

**Dinesh Jagai**
Department of Computer Science
University of Pennsylvania
Philadelphia, PA
dinesh97@seas.upenn.edu

**Thomas Wang**
Department of Computer Science
University of Pennsylvania
Philadelphia, PA
ttwang@seas.upenn.edu

Monday August $3^{rd}$ 2020

## 1 Overview

## 2 Concepts

1. The Potential field controller is formed by modeling the robot, as a particle under the influence of of a artificial potential field **U**. Such that **U** S superimposes i) Repulsive forces from obstacles and ii) Attractive force from goal.

   Now, the potential function is given by:

   $$U(q) = U_{att}(q) + U_{rep}(q) \quad \dots (1)$$

   And the Net force is given by :

   $$F_{net}(q) = \nabla U(q) \quad \dots (2)$$

   For this case, we took the mobile robot to be a particle, i.e. single fixed control point and used $U_{rep}$ as follows:

   $$U_{rep}(q) = \begin{cases} \frac{1}{2}\eta\left(\frac{1}{\rho(o(q))} - \frac{1}{\rho_0}\right)^2 & \rho(o(q) \leq p_0 \\ 0 & \rho(o(q) > p_0 \end{cases} \quad \dots (3)$$

   And subsequently, $F_{rep}$ is given by:

   $$F_{rep}(q) = \eta\left(\frac{1}{\rho(o(q))} - \frac{1}{\rho_0}\right)\left(\frac{1}{\rho^2(o(q))}\right)\nabla\rho(o(q)) \quad \dots (4)$$

   Where $\rho_0$ is the the distance of influence of an obstacle and $\rho(o(q))$ is the shortest distance between $o$ and any work-space obstacle

   For this assignment, we used $\rho_0$ as infinity so $\frac{1}{p_0}$ is 0

   Also, we assumed that the obstacle regions were convex and evaluated $\nabla\rho(o(q))$ to be

   $$\nabla\rho(o(q)) = \frac{o(q) - b}{||o(q) - b||}$$

   where $b$ is the point on the obstacle that is closest to $o(q)$

   Now, for the attractive force, we defined our potential field of attraction as follows:

   $$U_{att}(q) = \frac{1}{2}\zeta||(o(q) - o(q_f))||^2 \quad \dots (5)$$

Where $o(q_f)$ is the position of the goal. Note, We didn't used the **parabolic well potential** as we assumed that that the field grows linearly with distance of the robot to the goal.
As such, the force of attraction is given by :

$$F_{att}(q) = -\zeta(o(q) - o(q_f)) \quad \dots(6)$$

That is, $F_{net}$

$$= F_{rep} + F_{att}$$

$$= \eta\left(\frac{1}{\rho(o(q))} - \frac{1}{\rho_0}\right)\left(\frac{1}{\rho^2(o(q))}\right)\nabla\rho(o(q)) - \zeta(o(q) - o(q_f)) \quad \dots(7)$$

2. **Pseudocode for the planning algorithm**
The overall algorithm essentially calculates the net force, $F_{net}$ exerted on the mobile robot (treating it as a particle/single control point) in the artificial potential field. After finding the net force, the wheel velocities are found - this is done by finding $f_t \ \& \ f_p$ in the in the robot's frame (multiplying by a rotation matrix about $\theta$) and then solving $Ax = F$ where $x$ is the velocities of the robot and $F$ is composed of $f_t$ and $f_p$ in the robot's frame. Where $A = \left(\begin{smallmatrix} \alpha & \alpha \\ \beta & -\beta \end{smallmatrix}\right)$ such that $\alpha$ and $\beta$ are constants proportional to the wheels' radii. Note that in this case, $\alpha$ and $\beta$ were both set to the wheels' radii $(\frac{0.23}{2})$

Now, a more detailed approach to find $f_{net}$ is as follows.
For the repulsive force, $F_{rep}$ we used equation $(4)$, in order to get $\rho(q)$ we essentially looped through all the obstacles and found the one that was closed to the mobile robot. We then saved this distance and obstacle's position and used it to calculate $F_{rep}$ where $\rho(q)$ is this min distance and $b$ is the min obstacle's position.
Now, for $F_{att}$ equation $(6)$ was essentially used.
The values of $\zeta And \eta$ were initially set to $1$ but upon running different simulations (start and end points) they were adjusted to be $9 \ \& \ 3$. When the If your repulsive force was too strong, the value of $\zeta$ was reduced and when it was too high it was increased!

## 3 Coding Assignment

The main modified functions are shown below!

Listing 1: potential_field.py

```
import numpy as np
from math import sin, cos, atan2



def Planner_update(x, start, goal, robot, obstacles):
    """
    This function receives the current robot state, the start and goal positions,
    and obstacle information. It computes the net force (composition of attractive
    force and repulsive forces) based on the potential field planner, and it calls
    the select_wheel_vel function to update the wheel velocity command.

    Inputs:
        x,         current robot state ([x, y, theta], np.array(3))
        start,     start position for one test ([x, y, theta], np.array(3))
        goal,      goal position for one test ([x, y, theta], np.array(3))
        robot,     the mobile robot class
        obstacles, obstacle positions returned by lidar scan ([x, y], np.array(n, 2),
                   n is the number of obstacle points)
    Outputs:
        u,         desired wheel velocities ([right_wheel_vel, left_wheel_vel], np.array(2))
    """

    # Initialize net force
    F_net = np.zeros(2)

    # STUDENT CODE STARTS HERE
    # Assume P_0 = inft

    # Find \rho(O_i(q)), i.e. the shortest distance o_i and any workspace obstacle
    #  Then determine the repulsive force and the attractive force.
    #  Find the sum and return it!

    eta = 3
    zeta = 9
    min_dist = np.inf
    for obs in obstacles:
        a = (x[0] - obs[0])**2 + (x[1] - obs[1])**2
        dist = np.sqrt(a)
        if (dist < min_dist):
            min_dist = dist
            clostest_obs = obs
    #  Calculate the gradient for F_rep
    pos = np.array([x[0], x[1]])
    resultant = pos - clostest_obs
    grad_f_rep = (resultant)/(np.linalg.norm(resultant))
    F_rep = eta* (1/min_dist - 0)*(1/(min_dist ** 2))*(grad_f_rep.T)

    #  Calculate the attractive force
    goal_pos = np.array([goal[0], goal[1]])
    F_att = -zeta*(pos - goal_pos)

    F_net = F_att + F_rep
    # Compute wheel velocities from the net force and the current state
    u = select_wheel_vel(F_net, x)
    return u


def select_wheel_vel(F, x):
```

```
    """
    This function receives the net force computed by the potential field, and the current
    robot state. It projects the force onto the robot frame, computes the wheel velocities,
    and returns the velocity command.
    Inputs:
        F,      net force vetor in the potential field, (np.array(2))
        x,      current robot state ([x, y, theta], np.array(3))

    Outputs:
        u,      desired wheel velocities ([right_wheel_vel, left_wheel_vel], np.array(2))
    """

    # Initialize wheel speed
    u = np.zeros(2)
    if F[0] == 0 and F[1] == 0:
        return u

    # STUDENT CODE STARTS HERE
    # convert forces to the robot frame to calculate the velocities

    theta = x[2]
    rotation_matrix = np.array([[np.cos(theta), np.sin(theta)],
                [np.sin(theta), -np.cos(theta)]])
    # F = [[f_t], [f_p]]
    f = rotation_matrix@F
    print(f)
    # These should scale by the wheel radii
    alpha = 0.23/2
    beta = 0.23/2
    A = np.array([[alpha, alpha],
                [beta, -beta]])
    u = np.linalg.inv(A)@f
    print(u)
    return u
```

All other modified code is included in the appendix.

# 4 Simulation

### 4.1 For Tests in Static Environment

1. I started off with an easy path. The start was [0,0], and the end was [0,-1]. It successfully reached the goal, and it didn't experience too many problems. It didn't go in a very straight line as there were times that it fluctuates to the sides, but in the end it got to the goal easily.

2. Then, I tested one where the start location is [0,0], end location is [2,3]. There is a desk in between the two points. The robot in the beginning was struggling to find a direction to move in, but after a few seconds it decided on going in the y direction first, and then the x direction. It successfully avoided the desk and arrived at the destination.

3. I then tested with [-2,1] going to [-5,-2]. It successfully completed this path without much trouble. This is a clearer path because it's going almost straight to the target without many obstacles in the middle. When it got far enough from the obstacles, it moved more straight and quickly.

4. I tried to start at [3,2] and go to [4,-3]. In the beginning, the robot was moving more of less in place, and after a few seconds, it starts to move towards the desk. It successfully moved between the desk legs and towards the target. However, it got stuck near the small table, at about [4,-1]. As we could see, it was pretty close to the target, but didn't make it over.

### 4.2 For Tests in Dynamic Environment with Dynamic obstacles

1. I first tested with start location of [-2, 1] and end location of [-5,-2]. The robot was able to reach the start without a problem. When the moving balls were near it, the robot had some twitches where it didn't know where to go, but after the ball moved away it was able to find the destination.

2. Then, I tested with [3, 2] going to [4,-3]. While the path doesn't have much stationary obstacles in it, the moving ball really stopped the robot from proceeding. When the ball was far away, the robot moved towards the goal, but when it came close by, the robot moved backwards. Therefore, it couldn't reach the goal.

3. I attempted a hard one where the start location is [3,3] and end location is [-3,-3]. It's essentially traveling across the circular trajectory of the ball. The robot couldn't find its way through because the goal was very far away and there were too many obstacles in the middle. The robot didn't move far away from the starting point before it's stuck.

# 5 Questions to think about

1. In the 4 tests that I did, the robot succeeded 3 out of 4 times. It shows that it has a pretty good success rate. The 3 that it passed are indeed easily courses than the fourth, and the fourth one was quite complicated and far away. This shows that the robot can succeed on simpler small courses and succeed less on complicated large ones.

2. The planner worked well in situations where there aren't too many obstacles. If there were only one big obstacle in the middle, the planner can find a way around it. But if there are a few small ones around it, it sometimes gets lost and stuck. Also, it works well when the target is close by. When it's far away, there's a higher than that it gets stuck in the middle.

3. We did not implement an approach to escape local minima. I think a potential way talked in class is to take random jumps. When the robot is the same approximate location for too long, it can take a random jump to get out of the minima and find the path again. Also, the robot can take bigger steps each time so that it can get across of the minima or out of it.

4. One should use potential field when it's in a dynamic situation or a situation where you don't know the whole map beforehand. It can adapt to moving obstacles or unexpected obstacles because after sensing the obstacle, it can calculate the repulsive force and adjust its path. For algorithms like A* algorithm, it really needs to know the whole map before starting, and it would be much slower if it has to be in a dynamic situation. That be said, if one knows the map beforehand, an algorithm like A* or RRT would be more accurate and efficient.

5. The robot didn't work as well in the dynamic situation, only succeeding 1/3 of the tests. I think more tuning could improve the performance, and also it could perform better on easier courses that are farther away from the obstacles.

6. When the robot is close to the obstacle, the repulsive force would be stronger, and thus the robot would likely to move away from it. There are still times that the robot collides with the obstacles, and that's due to either the target is too attractive in the same direction as the obstacle such that the net force is still moving into the space of the obstacle. It could also be that there is too much repulsive force from other obstacles that push the robot towards this obstacle.
7. I changed the eta parameter so that the obstacles have a bigger repulsive force so that the robot can recognize the moving ball better. This didn't work too well because the robot just doesn't move towards the target anymore. Then, I increased the zeta parameter so that the target is more recognized. This made a difference and was able to guide the robot correctly in cases where there are less obstacles and it's an easier path.

# 6 Appendix

Listing 2: lidar_scan.py

```
from math import cos, sin
import numpy as np
import matplotlib.pyplot as plt

def lidar_scan(robot, x, max_dis = 5):
    """
    This function gets one scan from LIDAR, and transfers the data into positions
    of the point cloud. It also detects whether collision happens.
    Inputs:
        robot,     the mobile robot class
        x,         current robot state ([x, y, theta], np.array(3))
        max_dis,   (optional) maximum distance of the scan range, default is 5 m

    Outputs:
        obstacles, obstacle positions returned by lidar scan ([x, y], np.array(n, 2),
                   n is the number of obstacle points)
    """

    # Get scan updates from lidar
    scan = robot.get_scan()
    n = scan.shape[0]

    # Uncomment the following lines to show the scan result in a plot
    robot.set_wheel_velocity([0, 0])
    ax = plt.subplot(111, projection='polar')
    ax.scatter(np.linspace(0, 2*np.pi, n), scan)
    ax.set_rlim([0, 5])
    plt.show()

    # Do not modify code below
    # Get robot dimension
    r = robot.get_params()['axle-width']/2

    # Calculate obstacle coordinates (x, y) and detect collision
    obs_list = []
    for i in range(n):
        if scan[i] < max_dis:
            theta = x[2] + 2 * np.pi * i/n + np.pi
            obs_x = x[0] + cos(theta)*scan[i]
            obs_y = x[1] + sin(theta)*scan[i]
            obs_list.append([obs_x, obs_y])
        if scan[i] < r + 0.05:
            print("Failure: robot collided with obstacles")
            raise ValueError

    # Reshape to a m*2 numpy array
    obstacles = np.array(obs_list)

    return obstacles
```

Listing 3: run_sim.py

```
"""
Potential Field Planner: Main Simulation
@author: meam520
"""

from mobile_robot import MobileRobot
from potential_field import Planner_update
from lidar_scan import lidar_scan

from time import sleep
import matplotlib.pyplot as plt
import numpy as np


if __name__=='__main__':

    # Simulation setup
    # Modify this part for your tests
    start =  [4, -3, 0]       # [x, y, theta], coordinates of start, default: [0, 0, 0]
    goal = [0 ,0, 0]     # [x, y, theta], coordinates of goal

    print("Setup complete")
    print("Start:", start)
    print("Goal:", goal)

    robot = MobileRobot(start)

    # Simulation loop
    Done = 0
    goal_x = goal[0]
    goal_y = goal[1]

    while not Done:
        try:

            # Retrieve current robot state
            x, x_dot = robot.get_state()
            current_x = x[0]
            current_y = x[1]
            print("Current position:", x)

            # Update Lidar scan for obstacles
            obstacles = lidar_scan(robot, x)

            # Plan the desired robot velocity using potential field
            wheel_vel = Planner_update(x, start, goal, robot, obstacles)

            # Set desired robot velocity
            robot.set_wheel_velocity(wheel_vel)
            # print("Wheel speed:", wheel_vel)

            # Modify the goal-reaching conditions if needed
            if abs(goal_x-current_x) <= 0.5 and abs(goal_y-current_y) <= 0.5:
                Done = 1

            sleep(0.01)

        except (KeyboardInterrupt, ValueError):
        #except Exception as e:
         #   print(e)
            break

    # Stop the robot
```

```
robot.stop()
```