ME 570: Robot Motion Planning

Homework 3 Report

By Cameron Cipriano

11/02/2021

Problem 1: Drawing and Collision Checking for Spheres

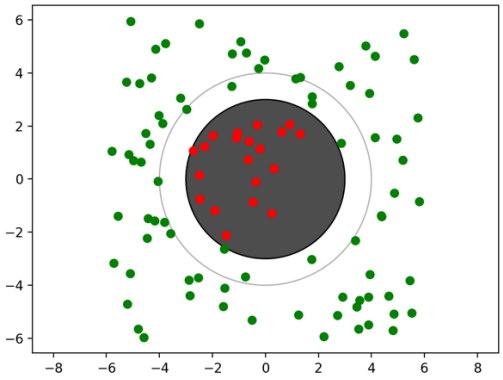
Question 1.1 code – Sphere.distance()

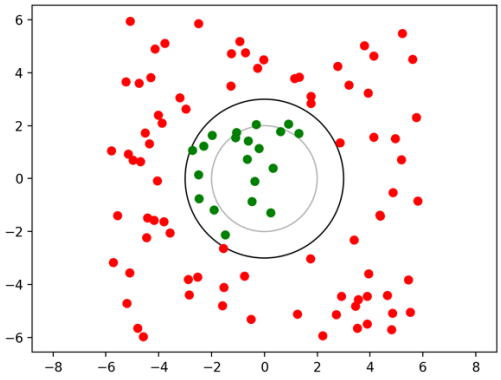
This function was just a matter of implementing $\|x - x_{center}\|_2 - r$ through np.linalg.norm()
The hollow sphere is the negative of the filled in verison

Question 1.2 code – Sphere.distance_grad()

The gradient of the distance function is $\nabla distance = \frac{x - x_{center}}{\|x - x_{center}\|}$. This too was implemented through np.linalg.norm(), taking the negative if the sphere was hollow

Question 1.1 optional – me570_hw3.sphere_testCollision()





Problem 2: The Potential-Based Planner

Question 2.1 code – RepulsiveSphere.eval()

Implemented Equation 3 from the PDF

Question 2.2 code – RepulsiveSphere.grad()

Implemented Equation 4 from the PDF utilizing the distance_gradient function

Question 2.3 code – Attractive.eval()

Implemented Equation 1 from the PDF

Question 2.4 code – Attractive.grad()

Implemented Equation 2 from the PDF

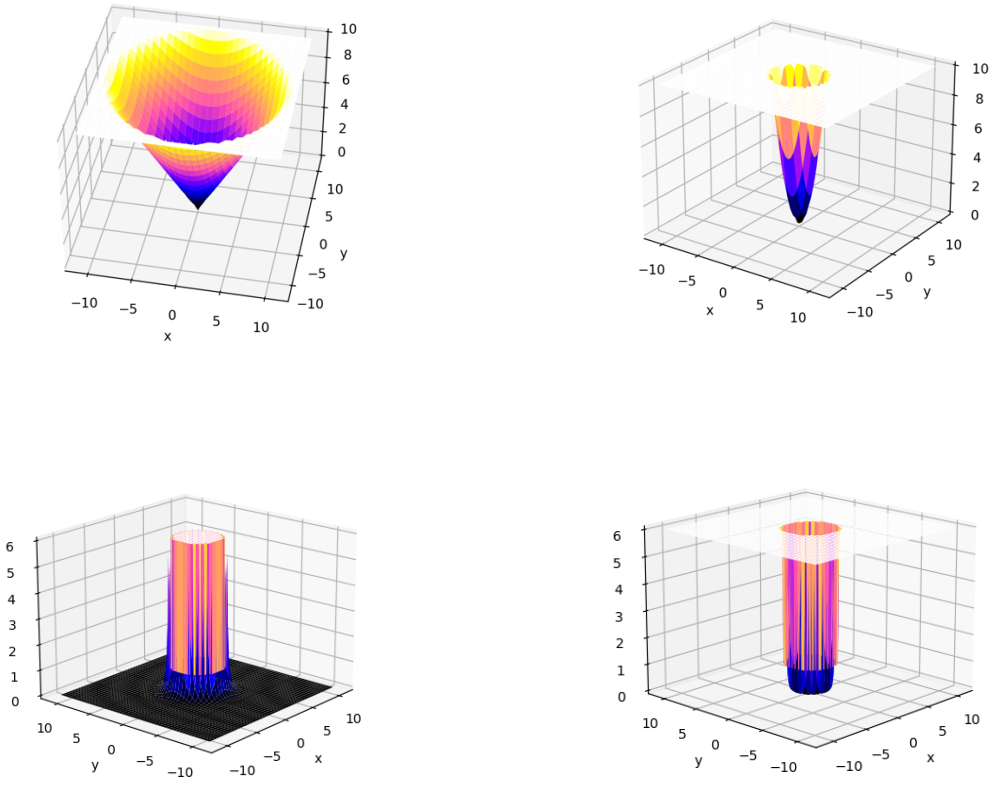
Question 2.5 code – Total.eval()

Implemented equation from the function's documentation on the assignment utilizing the **eval()** functions for the Attractive and RepulsiveSphere classes.

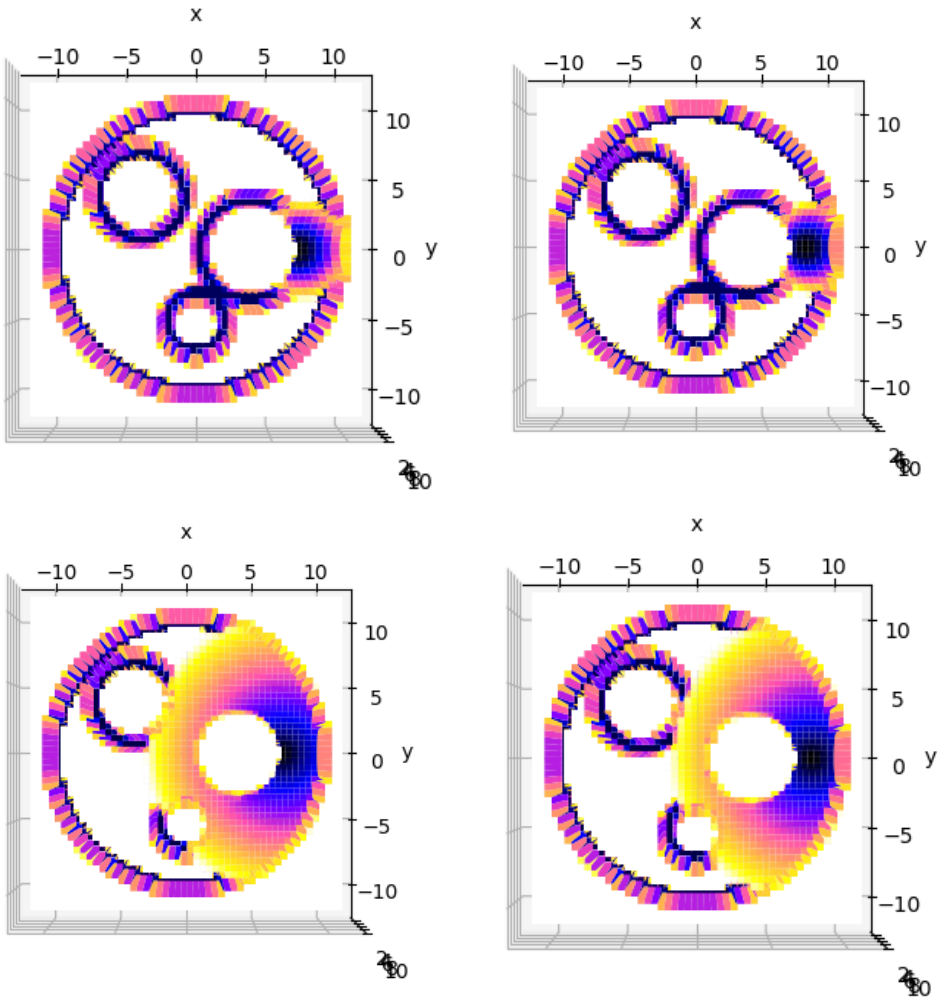
Question 2.5 code – Total.grad()

Implemented equation from the function's documentation on the assignment utilizing the **grad()** functions for the Attractive and RepulsiveSphere classes.

Question 2.1 optional – Using field_plotThreshold() to Visualize Attractive and Repulsive Potentials



Top Left: conic attractive potential; Top Right: quadratic attractive potential; Bottom Left: repulsive potential, solid sphere; Bottom Right: repulsive potential, hollow sphere

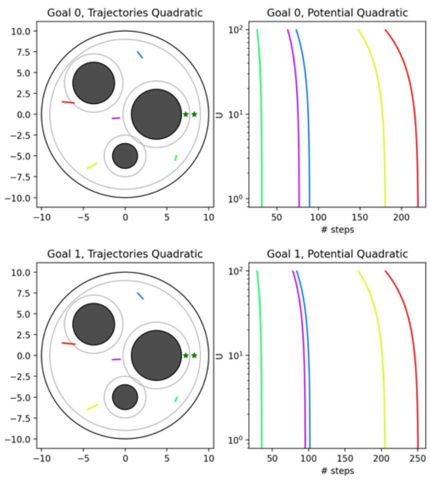


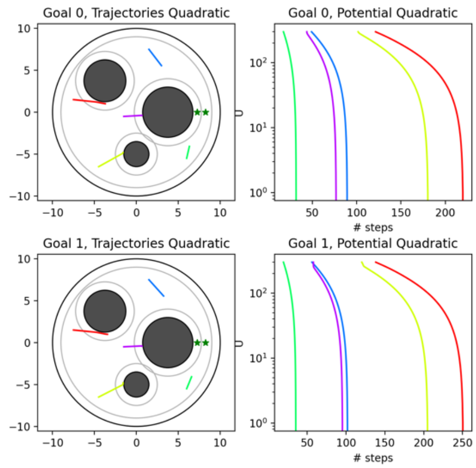
Top Left: total potential, quadratic goal 1; Top Right: total potential, quadratic goal 2; Bottom Left: total potential, conic goal 1; Bottom Right: total potential, conic goal 2

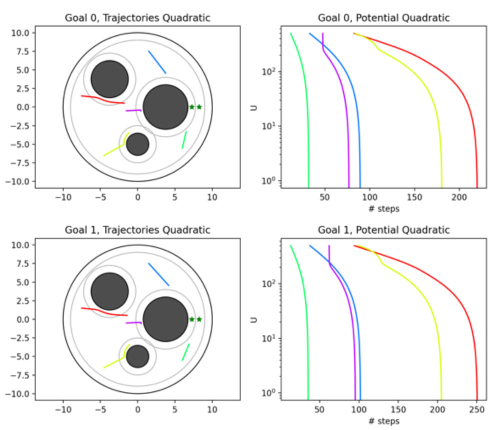
Question 2.6 code – Planner.run()

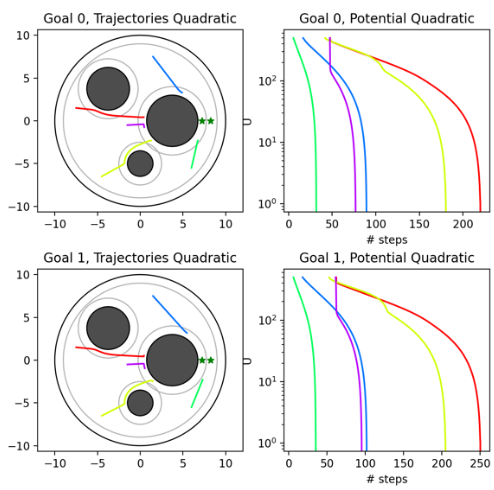
Question 2.7 code – Planner.run_plot()

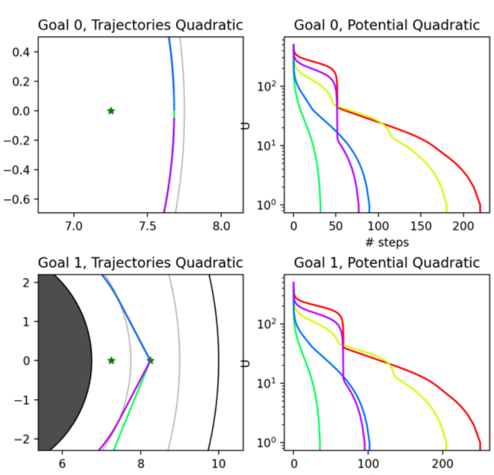
Question 2.1 report – Results from Planner.run_plot()

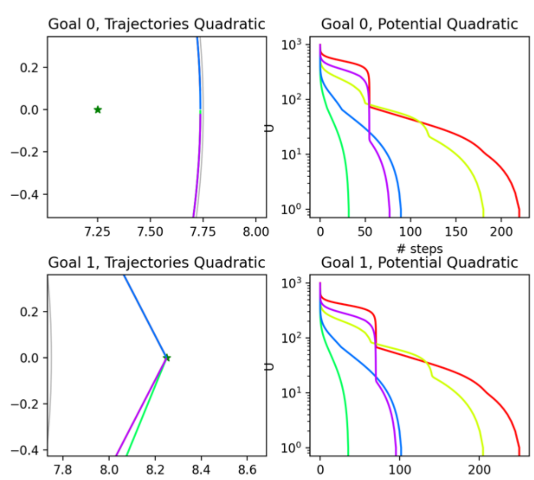






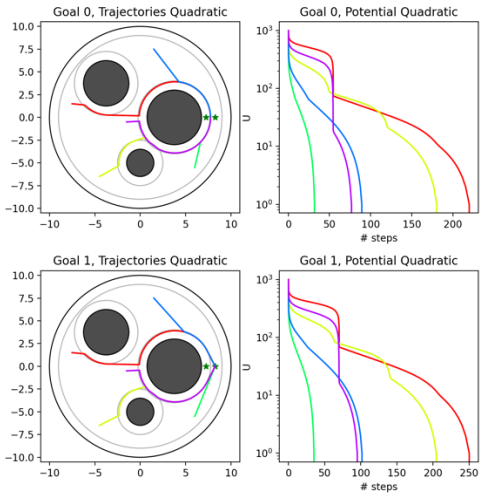




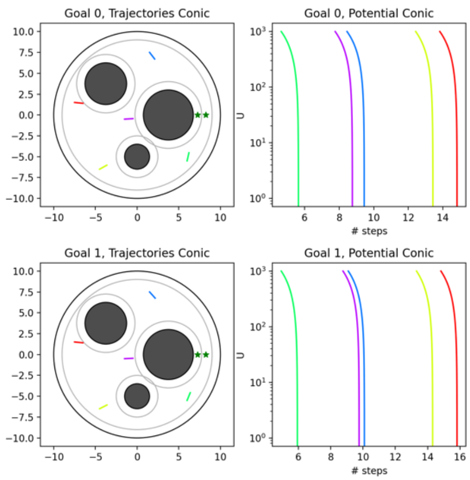


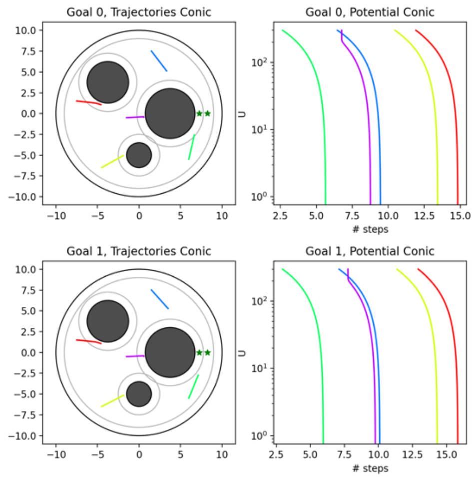
All above plots are for the quadratic attractive potential. First few plots range keep epsilon between 1e-3 and 1.75e-3, repulsive_weight between 0.01 and 0.5, and nb_steps between 100-500. We can see that these are not even coming close to converging on the goal, even with a significant increase in nb_steps, so I first increased the size of the epsilon, which affects how much influence the gradient has on the robot's position. Changing this to 1e-2 caused the robot at each starting location to converge quite closely to the goal.

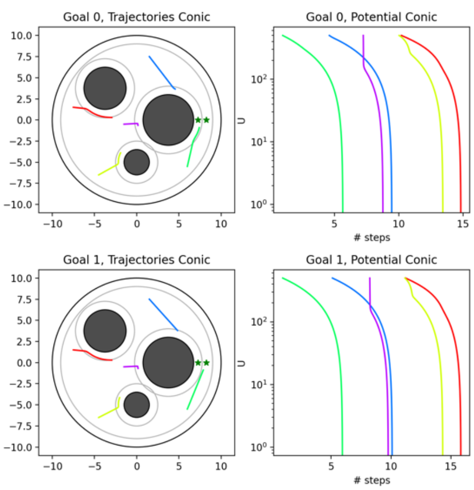
I noticed that the robot was going heavily into the distance of influence, so I then increased the repulsive weight. I found that the robot most closely followed the outline and avoided the obstacles when this weight was about 35. I am not sure if this is correct considering the high value, but it seems to be performing the task properly. When the repulsive weight is very high, only then will the robot fling out to infinity. The quadratic is much more sensitive to the epsilon value as it controls how the gradient affects its movement. Because the conic has the potential to overshoot the goal/ram further into the repulsive potential, it has greater chance to fling out to infinity.

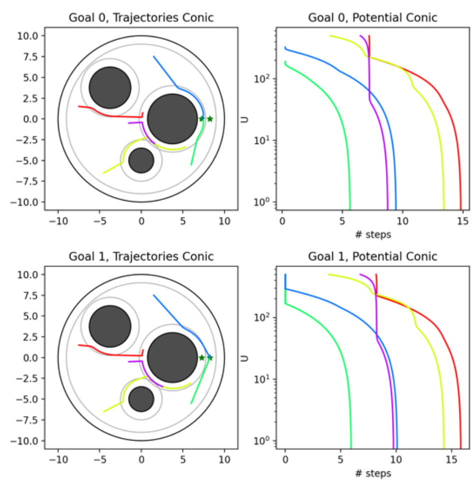


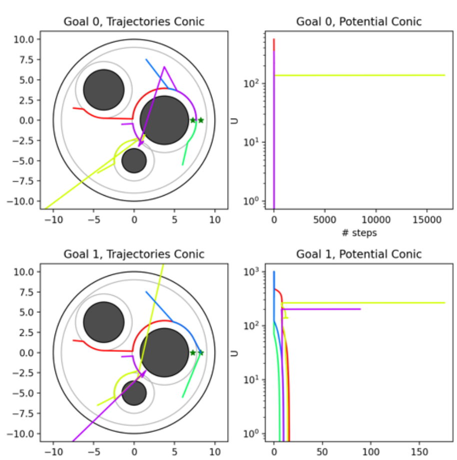
All below plots are for the conic attractive potential function. From what I can tell, the conic is much more sensitive to changes in the repulsive_weight than epsilon. Epsilon has to be much larger to make more of an effect towards the goal, as well as more iterations. When epsilon is very large, we observe an oscillating effect. When the repulsive_weight is very large, we observe the robot flinging off to infinity. The values I got for more reliable convergence were repulsive_weight of 1, epsilon at 3.55e-2, and nb_steps around 1000.

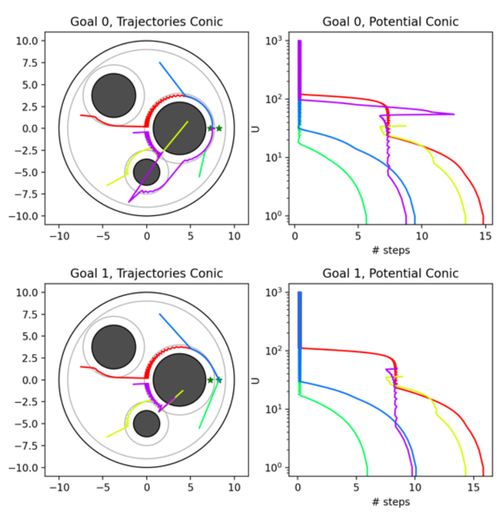

Goal 0, Trajectories Conic
Goal 0, Potential Conic
U
steps
Goal 1, Trajectories Conic
Goal 1, Potential Conic
U
steps

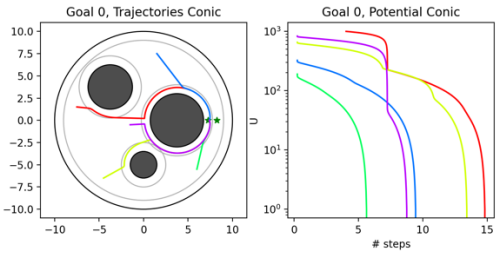

Goal 0, Trajectories Conic
Goal 0, Potential Conic
U
steps
Goal 1, Trajectories Conic
Goal 1, Potential Conic
U
steps

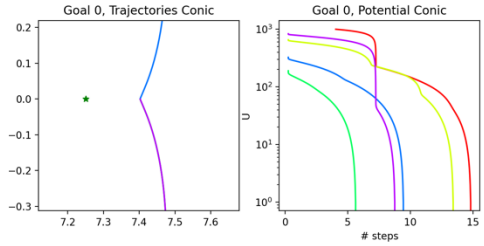

Goal 0, Trajectories Conic
Goal 0, Potential Conic
U
steps
Goal 1, Trajectories Conic
Goal 1, Potential Conic
U
steps

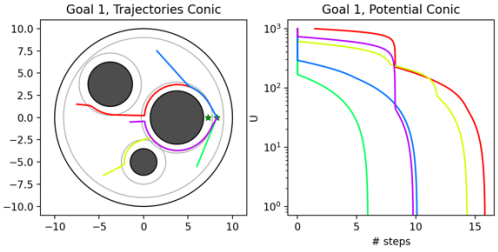

Goal 0, Trajectories Conic
Goal 0, Potential Conic
U
steps
Goal 1, Trajectories Conic
Goal 1, Potential Conic
U
steps

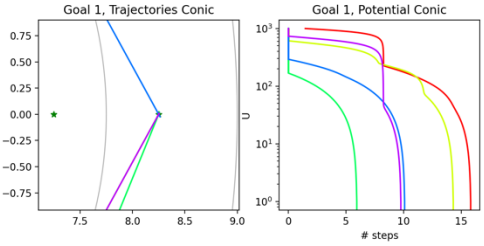

Goal 0, Trajectories Conic
Goal 0, Potential Conic
U
steps
Goal 1, Trajectories Conic
Goal 1, Potential Conic
U
steps


Goal 0, Trajectories Conic
Goal 0, Potential Conic
U
steps
Goal 1, Trajectories Conic
Goal 1, Potential Conic
U
steps





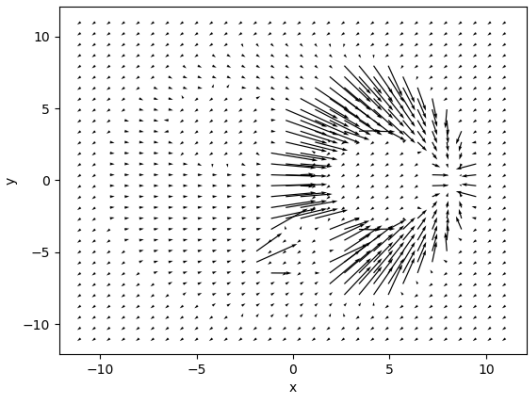


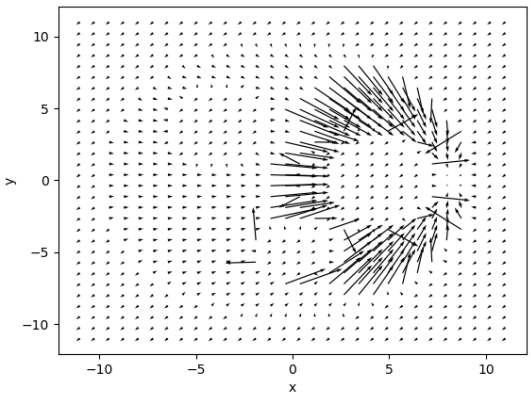


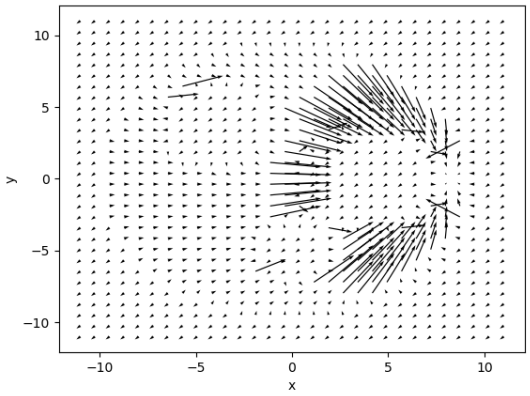
IMPORTANT NOTE
EACH PLOT ABOVE FOR THE POTENTIAL HAS THE X AND Y AXIS FLIPPED… I FIXED IT IN MY CODE SO THE NEW PLOTS WILL COME OUT CORRECTLY… MY APOLOGIES FOR NOT NOTICING IT EARLIER

Question 2.2 report – Visualizing the total potential, U, and its gradient ∇U

Total Potential can be seen in the above graphs from Question 2.1 Optional. The gradient can be visualized below with repulsive weights 0.5, 5, and 35 respectively







Question 2.3 report – Effects of varying repulsive weight and epsilon

Discussion is above in report 2.1.

Question 2.4 report – Relation between the value of U toward the final iterations and its effect on the planner's success/failure.

The value of U should be monotonically decreasing all the way until the end. Towards the final iterations, the change in U should be smaller and smaller as it approaches 0. If the planner is not going to succeed, we would see an asymptotically increasing value as the robot shoots off to infinity if the values of epsilon and the repulsive_weights are not tuned correctly.

Question 2.5 report – Difference between two goal positions

The only difference between the goal positions is that one is inside the distance of influence for the obstacle. Depending on how strong the repulsive weight is, the closer or farther the robot will be able to reach the goal. If the

Problem 3: CLF-CBF Formulation

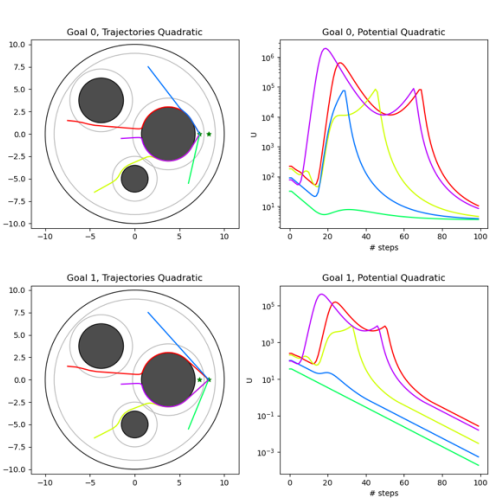
Question 3.1 report – Write the expressions for the clover involving U_{attr} and the spade involving $d_i(x)$

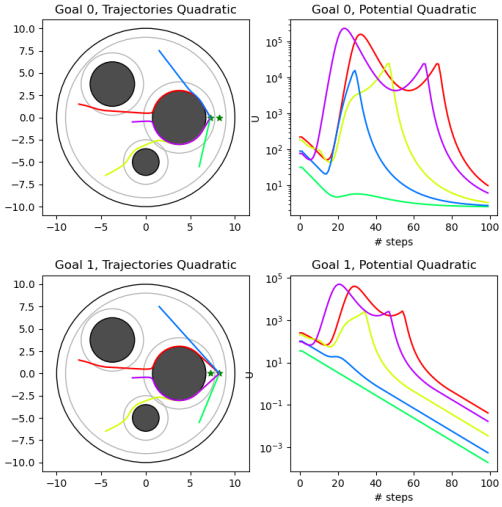
Clover: $-\nabla U_{attr}$ → We are using this as the reference controller as this is the direction we are always trying to head.

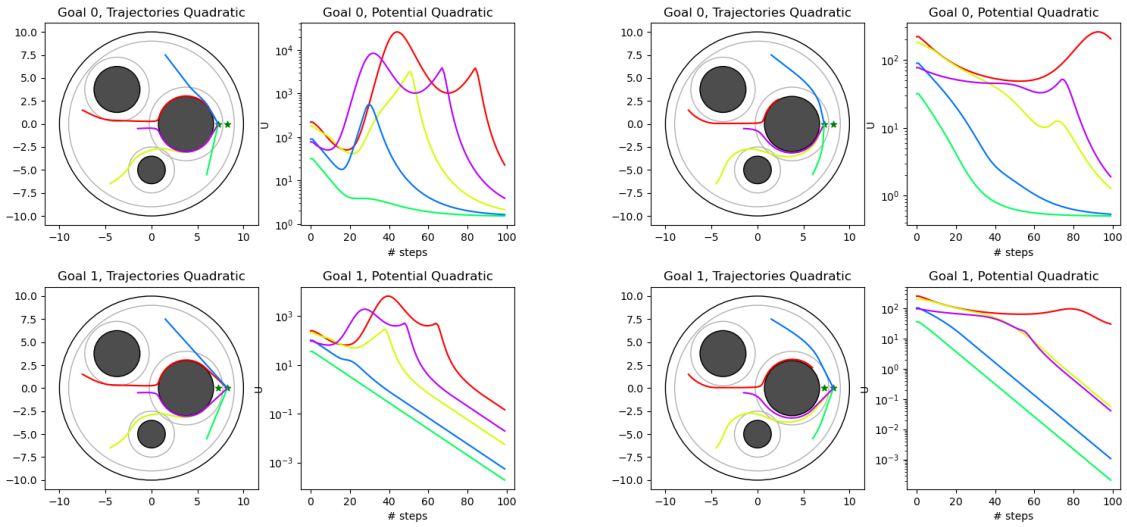
Spade: $-\nabla d_i(x)^T u - \alpha \cdot d_i(x) \leq 0$

Question 3.1 code – me570_potential.clfcbf_control()

Question 3.2 report – Results from potential_planner_runPlot()

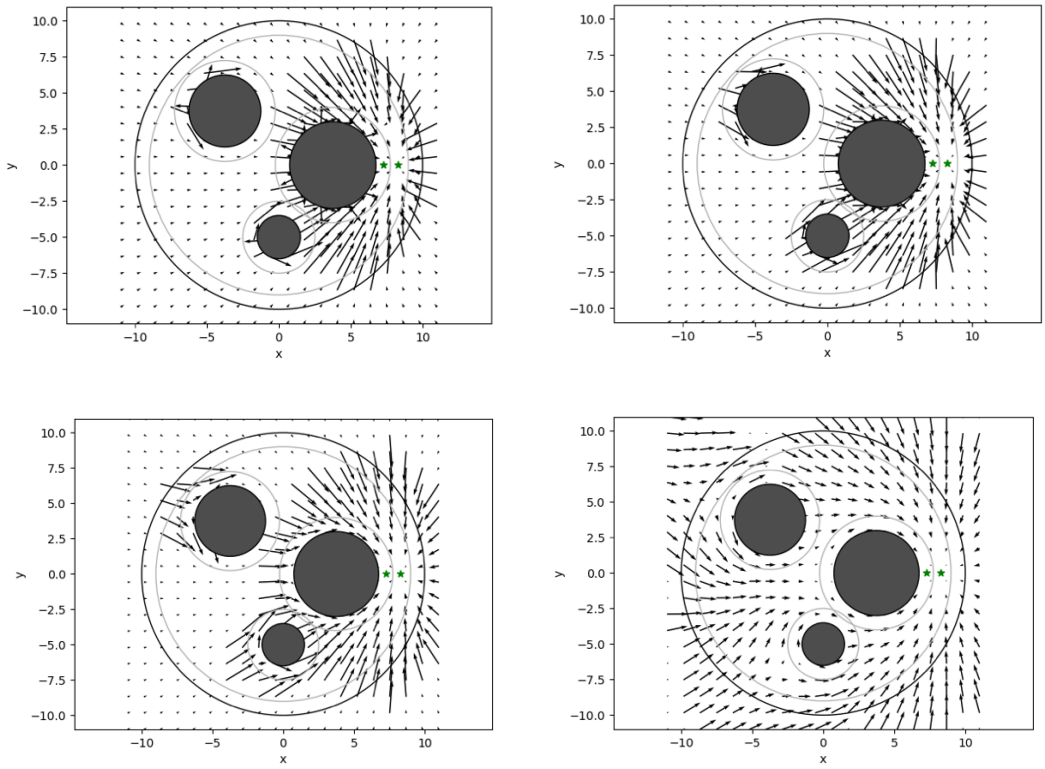






Repulsive Weights from first to last: 7, 5, 3, 1
Epsilon: 6.0e-2
Nb_steps: 100

Question 3.3 report – Results from field_plotThreshold() to visualize the control $u^*(x)$ for each combination of repulsive weights



Question 3.4 report – Trade-off between traditional gradient-based methods and a CLF-CBF formulation

The main trade-off I see between the gradient-based method and a CLF-CBF formulation is the avoidance of obstacles. The gradient-based method does a better job of steering clear of obstacles through the repulsive

Problem 4: Jacobian-Based Inverse Kinematics for the Two-Link Manipulator

Question 4.1 report – Write an expression for the Jacobian Matrix J such that $\frac{d}{dt}\left({}^{w}p_{eff}\right) = J(\theta)\dot{\theta}$

$$ {}^{w}p_{eff} = \begin{bmatrix} 5(\cos(\theta_1)\cos(\theta_2) - \sin(\theta_1)\sin(\theta_2)) + 5\cos(\theta_1) \\ 5(\sin(\theta_1)\cos(\theta_2) + \cos(\theta_1)\sin(\theta_2)) + 5\sin(\theta_1) \end{bmatrix} $$

Considering the end-effector is located at ${}^{\beta_2}p_{eff} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$

$$ J(\theta) = \begin{bmatrix} 5(-\sin(\theta_1)\cos(\theta_2) - \cos(\theta_1)\sin(\theta_2)) - 5\sin(\theta_1) & 5(-\cos(\theta_1)\sin(\theta_2) - \sin(\theta_1)\cos(\theta_2)) + 5\cos(\theta_1) \\ 5(\cos(\theta_1)\cos(\theta_2) - \sin(\theta_1)\sin(\theta_2)) + 5\cos(\theta_1) & 5(-\sin(\theta_1)\sin(\theta_2) + \cos(\theta_1)\cos(\theta_2)) + 5\sin(\theta_1) \end{bmatrix} \dot{\theta} $$

Question 4.1 code – TwoLink.jacobian_matrix()

Question 4.1 optional – Compare Jtheta*thetaDot with the results of TwoLink.jacobian(theta, thetaDot)

The two produce the same output for the same inputs for $\theta = \begin{bmatrix} \pi/2 \\ \pi/2 \end{bmatrix}$ and $\dot{\theta} = \begin{bmatrix} \pi/4 \\ \pi/4 \end{bmatrix}$

```
In [100]: two_link.jacobian_matrix(theta) @ theta_dot
Out[100]:
array([[-3.92699082],
       [-3.92699082]])

In [101]: two_link.jacobian(theta, theta_dot)
Out[101]:
array([[-3.92699082],
       [-3.92699082]])
```

Question 4.2 code – TwoLinkPotential.eval()

Question 4.2 code – TwoLinkPotential.grad()

Question 4.3 code – TwoLinkPotential.plot_animate()

Unfortunately, I did not got to this

Question 4.4 code – TwoLinkPotential.run_plot()

Unfortunately, I did not got to this

Question 4.1 report – Results from twolink_planner_runPlot()

Unfortunately, I did not got to this