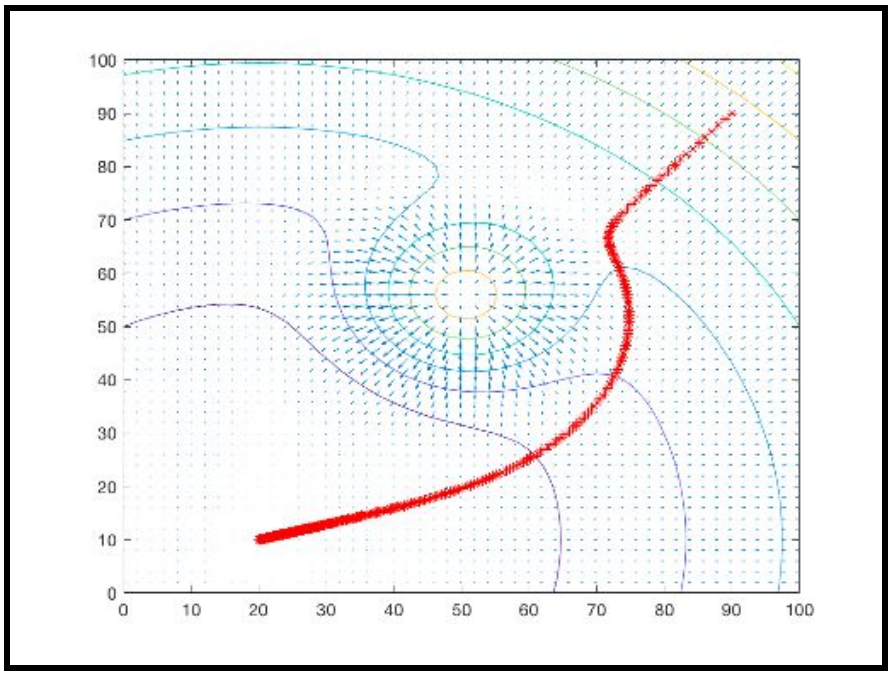
Gaotong Wu
A13809639
ECE 172A Hw2

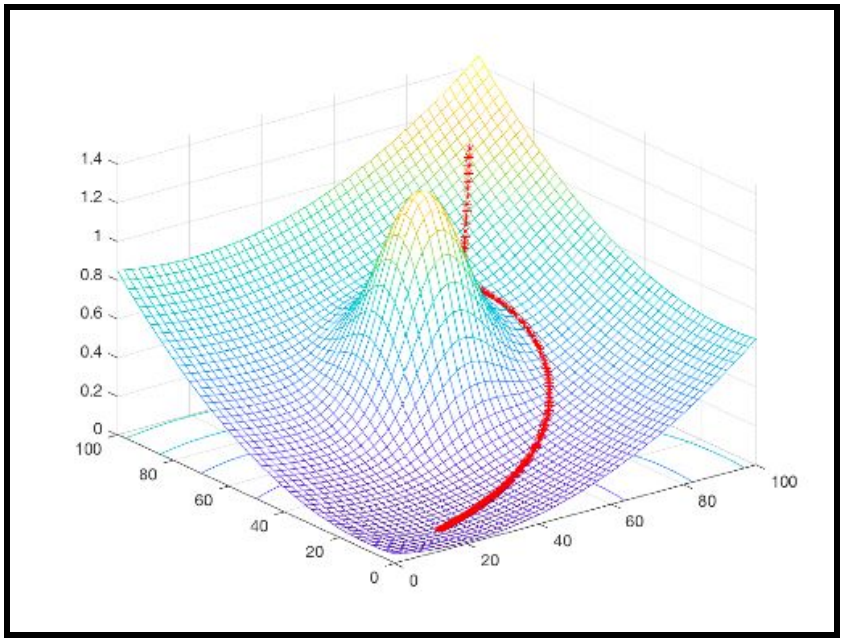
Academic Integrity Policy: Integrity of scholarship is essential for an academic community. The University expects that both faculty and students will honor this principle and in so doing protect the validity of University intellectual work. For students, this means that all academic work will be done by the individual to whom it is assigned, without unauthorized aid of any kind.

By including this in my report, I agree to abide by the Academic Integrity Policy mentioned above.

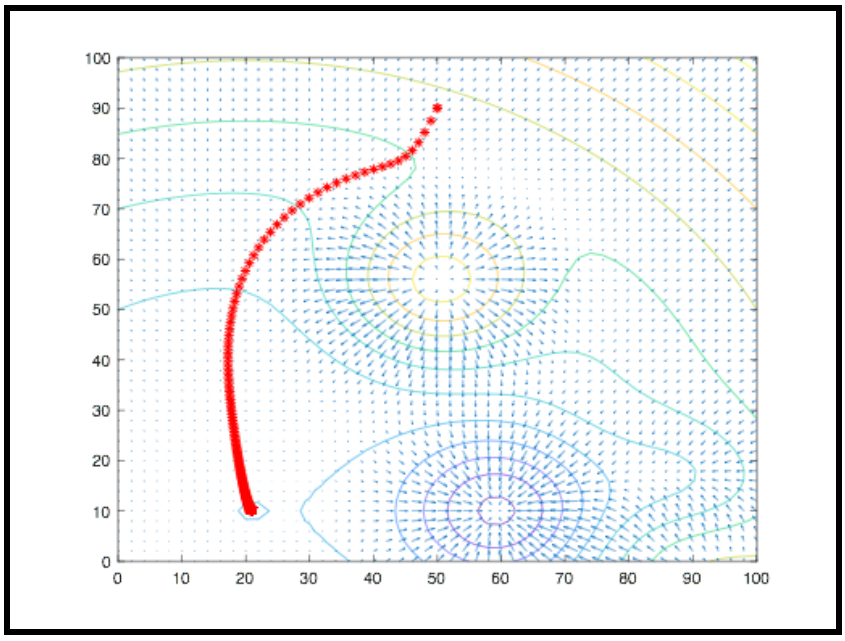
Problem 1



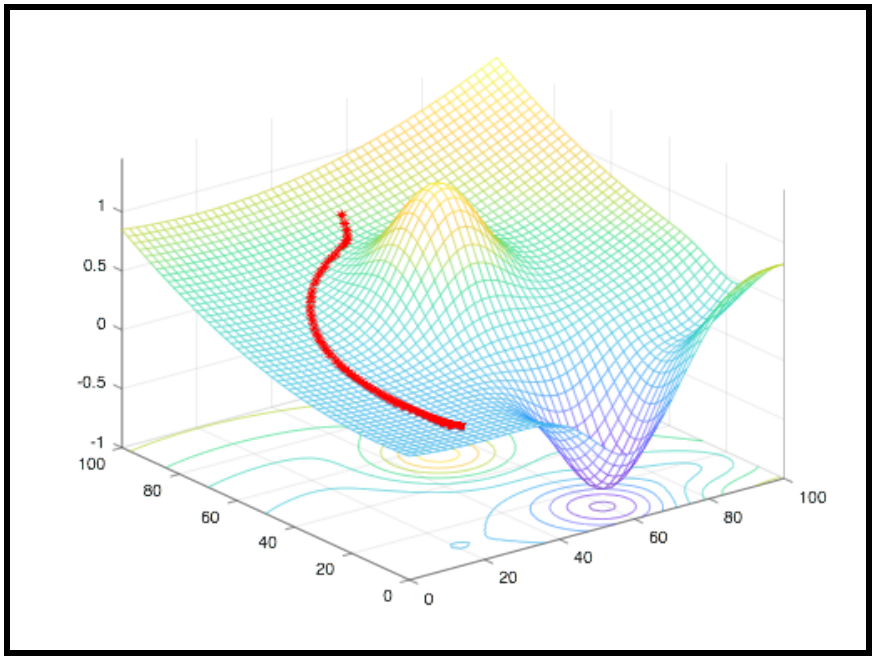
Contour and quiver plot for the first layout



Gradient Descent of the first layout



Contour and quiver plot for the second layout



Gradient Descent for the second layout

(ii)

- **Why do the obstacles look like they do on the potential field?**
  In the mesh plot, the obstacles are modeled by repulsors which will push the bot away from it. On the quiver/contour plot, the obstacles have negative gradient pointing upwards; when we implement the gradient descent algorithm, the bot will get away from them and move to a lower gradient position.

- **What happens to the gradient as you approach the end location?**
  The gradient approaches 0
  The gradient approaches to 0.
- **Why are we plotting the negative gradient instead of the gradient?**

  Since we are implementing gradient descent algorithm, the negative gradient shows us how the bot will move.

(iii)

- **Why is this method better than the sense-act paradigm?**

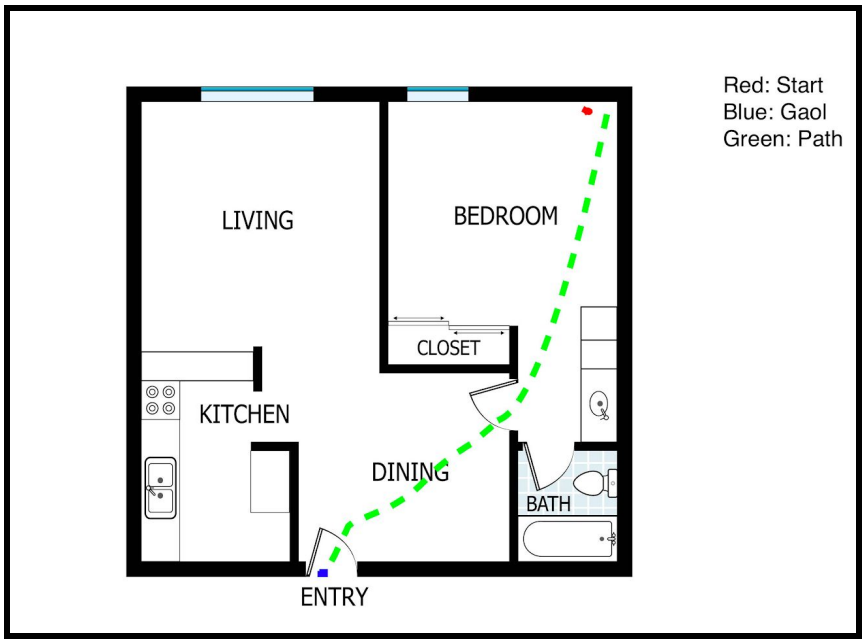
  Gradient descent algorithm can be more widely applicable to many other environments. Specifically, it models obstacles as repulsors and places you would like to head to as attractors. This allows the bot to go to any preferred destinations, and the bots can transverse any directions. However, sense-act paradigm only allows bot heading downwards and moving along non-diagonal directions.

- **Explain what the algorithm is doing and why it works to get the bot across the room while avoiding obstacles?**

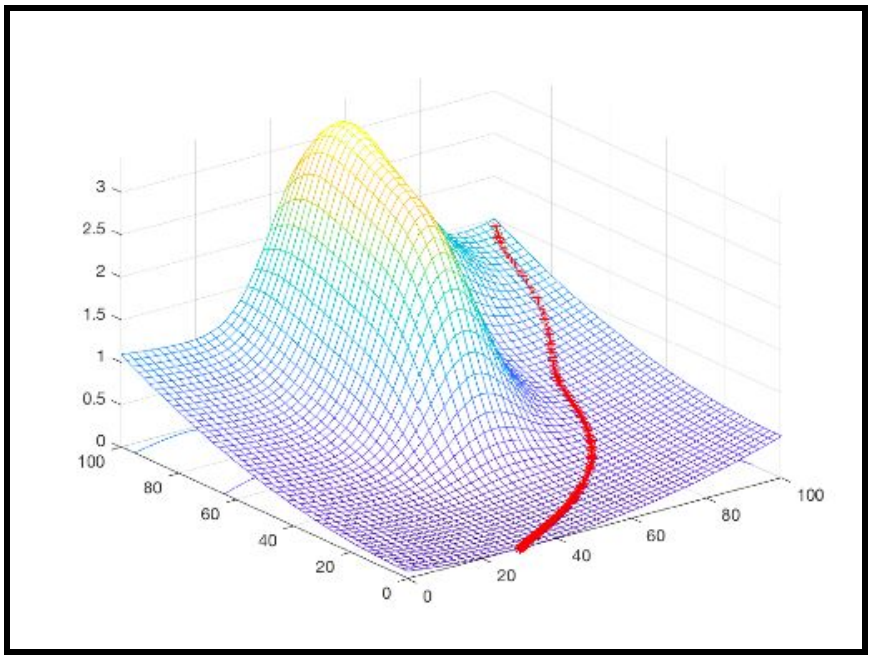
  The gradient descent algorithm allows the bot to move in the negative gradient directions until the gradient approaches zero. In the potential field, obstacles are modeled as upwards negative gradient and the goal is model as zero gradient; when implementing gradient descent, the bot will move away from the high gradient (obstacles) and towards zero gradient (goal). That's how the bot moves to the goal with colliding any obstacles.

Extra Credit

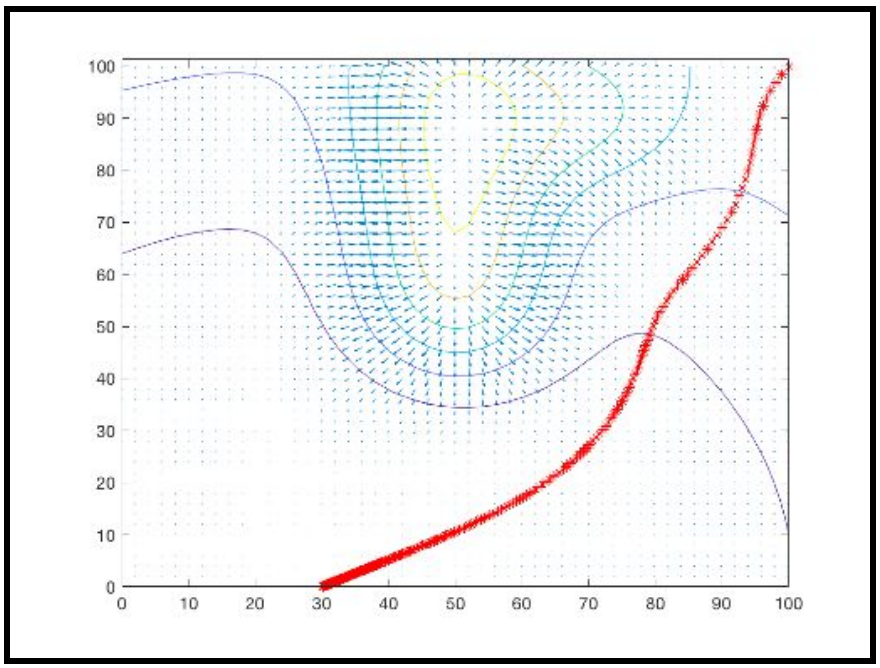
(a)



Floor Plan I chose

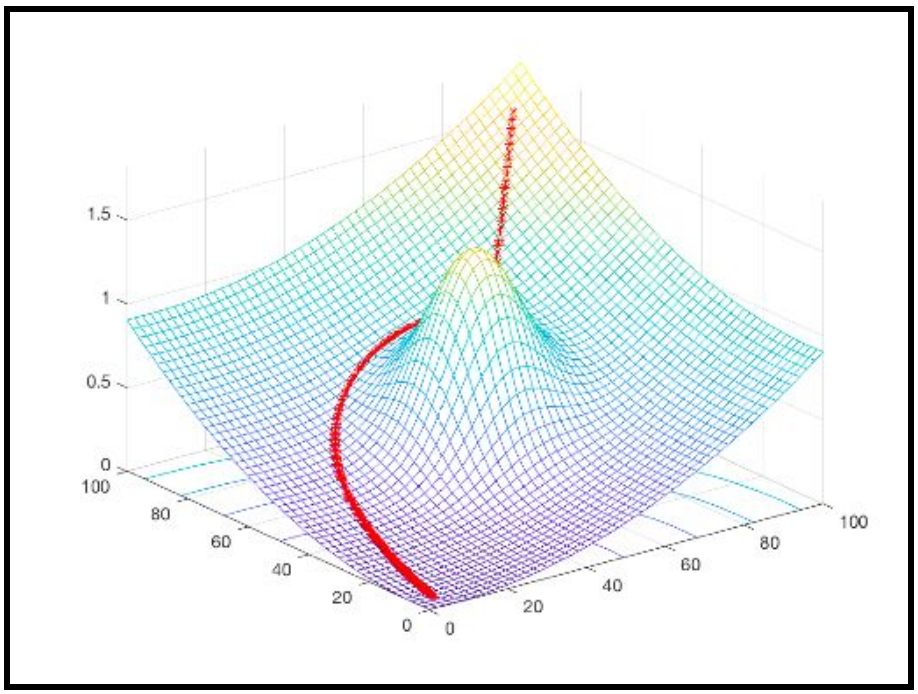


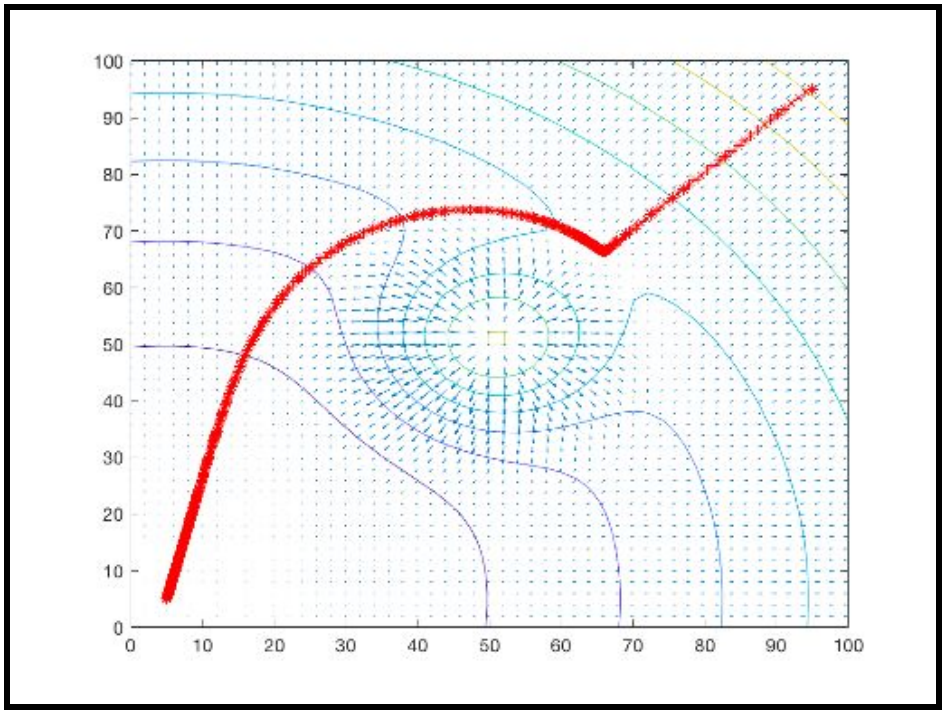
Mesh plot of the floor plan



contour/quiver plot of the floor plan

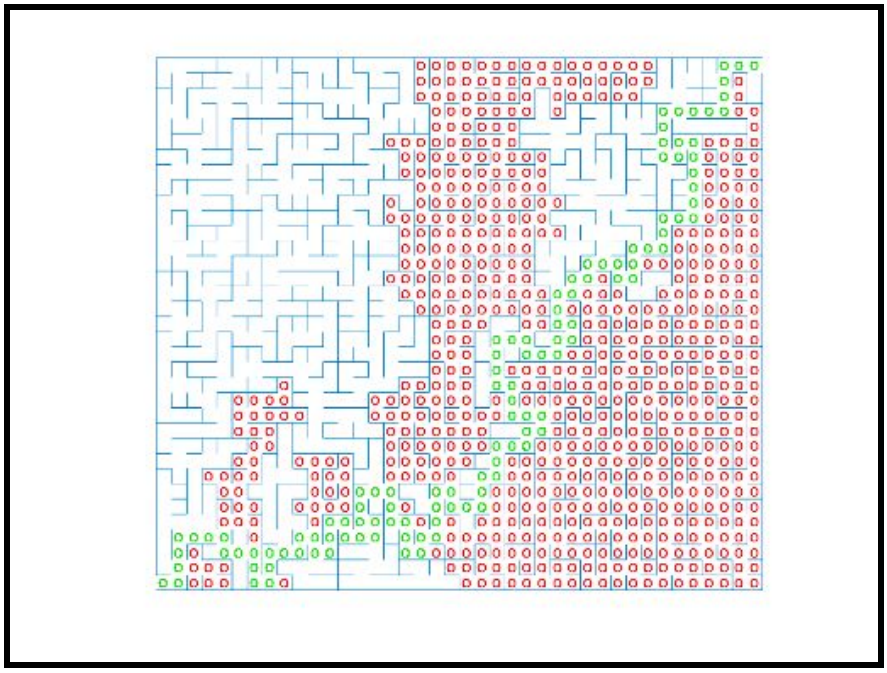
(b)





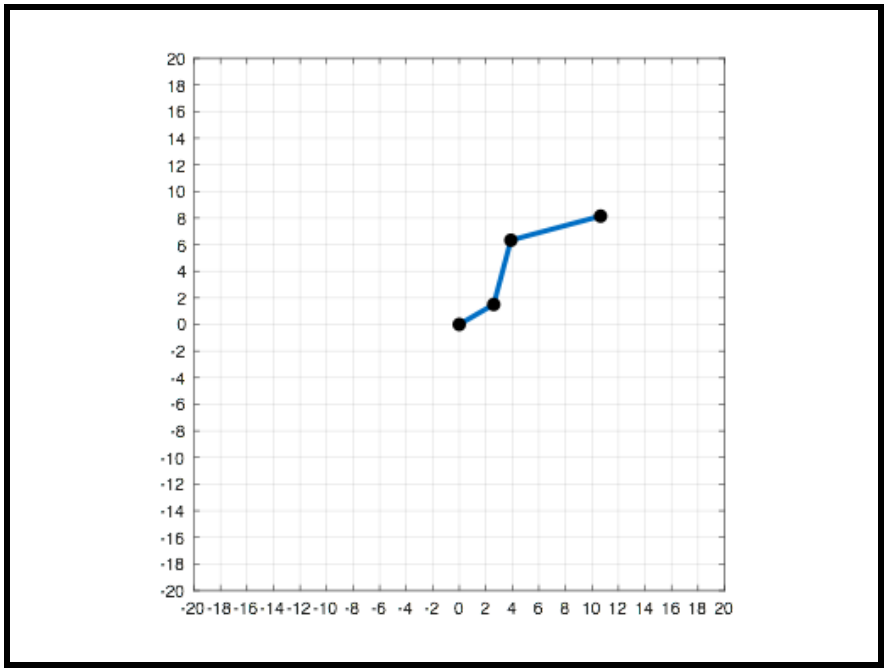
The original gradient descent is not able to reach the goal. It's because the bot hits the saddle point where the gradient is zero but not the goal yet. In order to solve this, the first gradient descent direction A_x cannot be on any points along the line formed by [5,5] and [95,95], which includes the initial position. Setting the first descent direction on any points not on this line is fine. Code to do this is on the Appendix below.

Problem 2

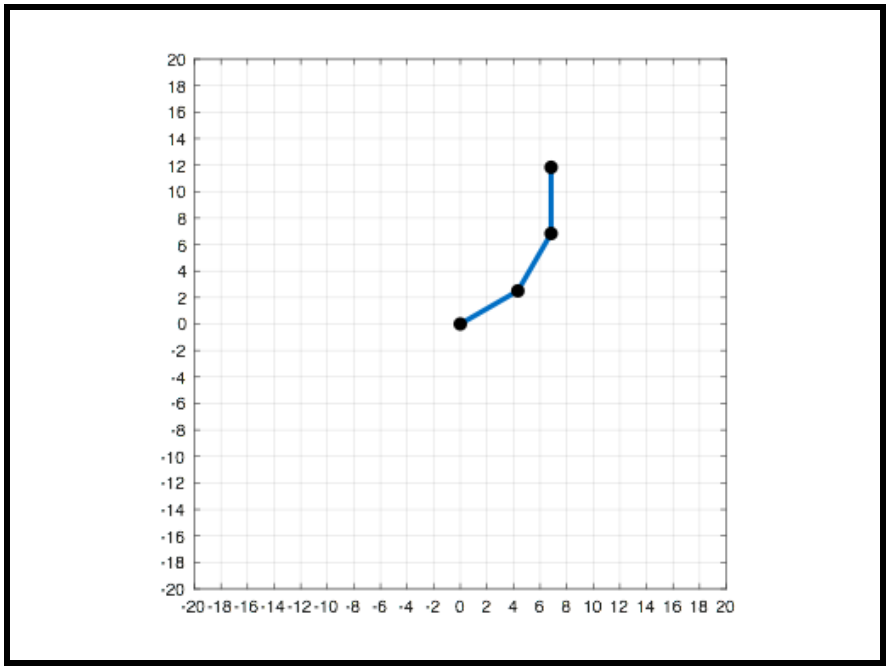


Problem 3

forwards kinematics

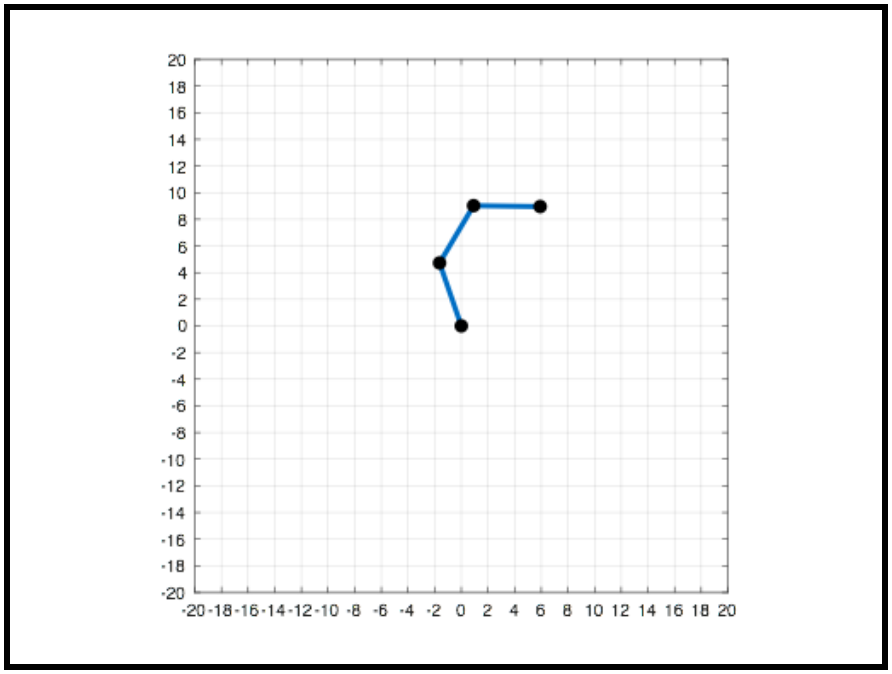


Initiation (a) forwards kinematics

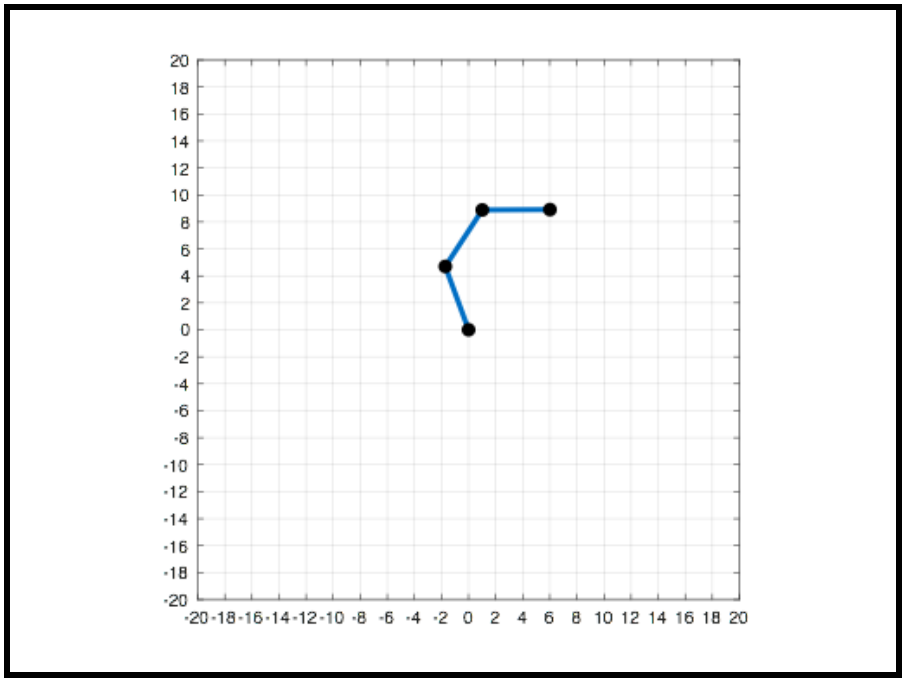


Initiation (b) forwards kinematics

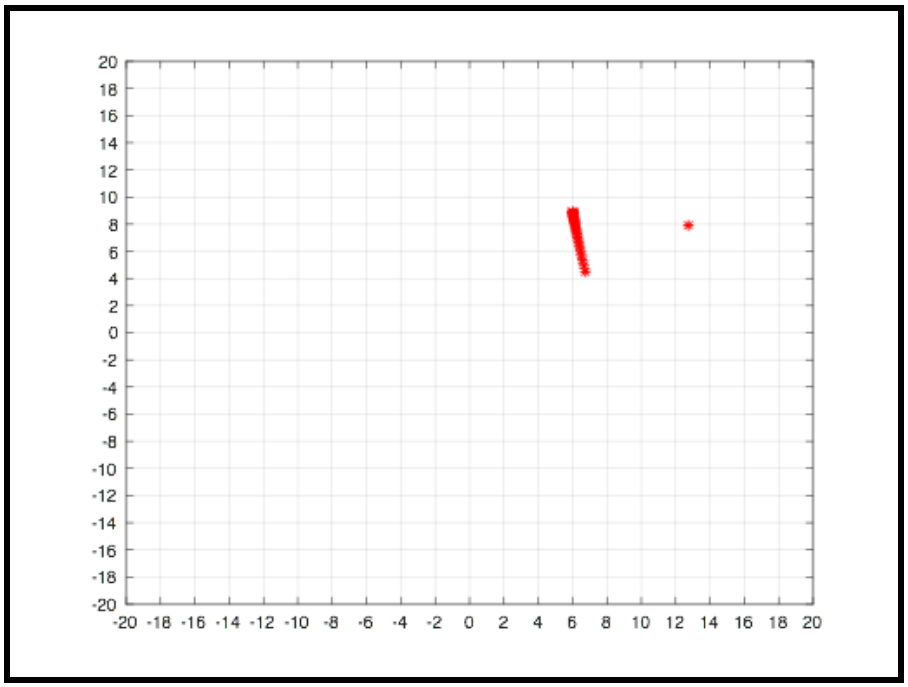
Inverse Kinematics



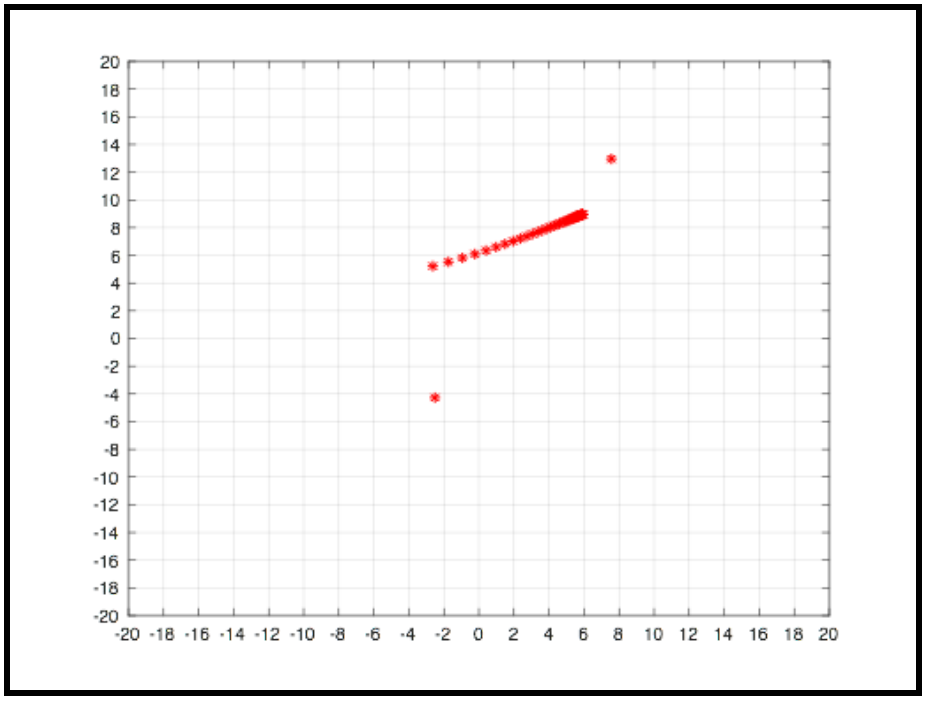
end effector position with initiation (1)



end effector position with initiation (2)



end effector position with initiation (1)



end effector position with initiation (2)

Problem 4

(i)(a) Bug 1 Algorithm

Lower Bound:

$$T \geq D$$
$$T \geq 220m$$

Upper Bound:

$$T \leq D + 1.5 * \Sigma p_i = 220m + 1.5 * (500m + 14 * 4m) = 1054m$$

(b)Bug 2 Algorithm

Lower Bound:

$$T \geq D$$
$$T \geq 220m$$

Upper Bound:

$$T \leq D + 0.5 * \Sigma n_i p_i = 220m + 0.5 * (2 * 500 + 14 * 4) = 748m$$

(ii)

Bug 0, Bug 1 and Bug 2 all will reach the endpoint.

Bug 0 would result in the least distance traveled because the bot will be directly heading to the endpoint once there are no obstacles along the line between the bot and the endpoint. Bug 1 circumvents all the obstacles to find the closest point to the endpoint and Bug 2 takes a longer path to get to the goal line when it hits an obstacle.

(iii)

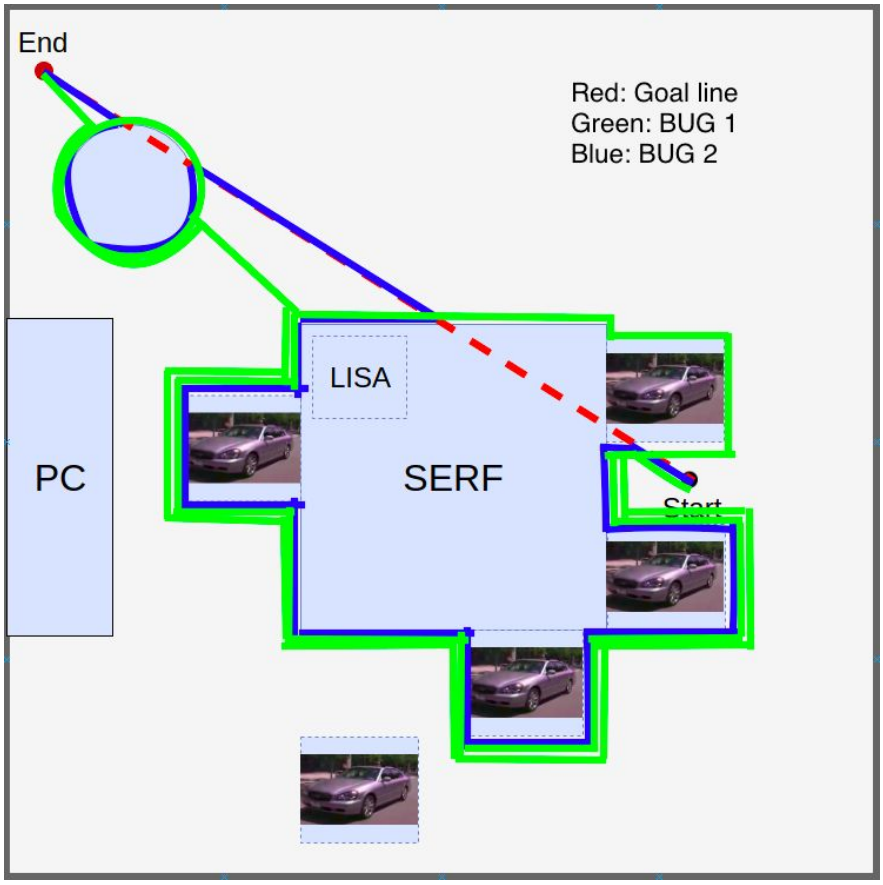
Bug 0, Bug 1 and Bug 2 all will reach the endpoint.

Bug 0 would result in the least distance traveled because the bot will be directly heading to the endpoint once there are no obstacles along the line between the bot and the endpoint. Bug 1 circumvents all the obstacles to find the closest point to the endpoint and Bug 2 takes a longer path to get to the goal line when it hits an obstacle.

(iV)

Bug 1 and Bug 2 will reach the endpoint.

Bug 2 would result in the least distance traveled. Bug 2 goes along the edge of the obstacles once it hits to find the goal line; when there are no obstacles along the goal line it directly heads to the endpoint. Bug 1 circumvents all the obstacles to find the closest point to the endpoint. Bug 0 fails to reach the goal because the bot will get stuck in the lower left corner between the two cars.


End
Red: Goal line
Green: BUG 1
Blue: BUG 2
PC
LISA
SERF
Start

Problem 1

```
% add different loc,goals,repulsors,attractors to the potential field
loc = [90, 90];
goal = [20, 10];
repulsor1=[50,55];
repulsor2=[90,5];
attractor1=[60,40];
attractor2=[20,100];
potentialField = Potential(100, 100);
potentialField=addGoal(potentialField,goal);
potentialField=addRepulsor(potentialField,repulsor1);
potentialField=addRepulsor(potentialField,repulsor2);
potentialField=addAttractor(potentialField,attractor1);
potentialField=addAttractor(potentialField,attractor2);

%Plot the contour and quiver of the potential field
figure();
contour(x,y,z(x,y));
hold on
plot(90,90,'r*');
figure();
quiver(x,y,-dx(x,y),-dy(x,y));
hold on
plot(90,90,'r*');

%Implement the gradient descent algorithm and plot the position of the bot
for each iteration
figure();
alpha=1/norm(dx(loc(1),loc(2)),dy(loc(1),loc(2)));
A_x=[dx(loc(1),loc(2)),dy(loc(1),loc(2))];
epsilon=10^-10;
meshc(x, y, z(x, y));
hold on
while norm(dx(loc(1),loc(2)),dy(loc(1),loc(2)))>epsilon
    loc=loc-alpha*A_x;
    A_x=[dx(loc(1),loc(2)),dy(loc(1),loc(2))];
    plot3(loc(1),loc(2),z(loc(1),loc(2)),'r*');
end
```

Extra Credit

```
A_x=[dx(40,50),dy(40,50)];% for the first gradient descent step set the
direction on any points that are not on the line [5,5],[95,95];
epsilon=10^-10;
meshc(x, y, z(x, y));
hold on
while norm(dx(loc(1),loc(2)),dy(loc(1),loc(2)))>epsilon
    loc=loc-alpha*A_x;
    A_x=[dx(loc(1),loc(2)),dy(loc(1),loc(2))];
    plot3(loc(1),loc(2),z(loc(1),loc(2)),'r*');
    plot(a1,loc(1),loc(2),'r*');
end
```

Problem 2

```
% Implement DFS
N=data.num_cols*data.num_rows;
stack=java.util.Stack();
visited=zeros(N,1);
parents=zeros(N,1);
stack.push(cur_loc);
while ~stack.isempty() && ~isequal(cur_loc,N)
     cur_loc=stack.pop();
    if visited(cur_loc)==0
        visited(cur_loc)=1;
        neighbors=sense_maze(cur_loc,data);
        for i=1:length(neighbors)
           if visited(neighbors(i))==0
               stack.push(neighbors(i));
               parents(neighbors(i))=cur_loc;
           end
        end
    end
    draw_cursor(cur_loc,[data.num_rows,data.num_cols],"r",h);
end
cur_loc=data.num_cols*data.num_rows;

%plot the final
while cur_loc~=0
    draw_cursor(cur_loc,[data.num_rows,data.num_cols],"g",h);
    cur_loc=parents(cur_loc);
end
```

Problem 3

```
%forwards kinematics
function [x_1,y_1,x_2,y_2,x_e,y_e] = ForwardKinematics(l0,l1,l2, theta0,
theta1, theta2)
x_1=l0*cos(theta0);
y_1=l0*sin(theta0);
x_2=x_1+l1*cos(theta0+theta1);
y_2=y_1+l1*sin(theta0+theta1);
x_e=x_2+l2*cos(theta0+theta1+theta2);
y_e=y_2+l2*sin(theta0+theta1+theta2);
end

%inverse kinematics

while  ((x_e-x_e_target)^2+(y_e-y_e_target)^2)^0.5>0.1 % Replace the '1'
with a condition that checks if your estimated [x_e,y_e] is close to
[x_e_target,y_e_target]

       %Calculate the Jacobian
        Jacobian=zeros(2,3);
        Jacobian(1,1)=-l0*sin(theta0)-l1*sin(theta0+theta1)-
l2*sin(theta0+theta1+theta2);
        Jacobian(1,2)=-l1*sin(theta0+theta1)-l2*sin(theta0+theta1+theta2);
        Jacobian(1,3)=-l2*sin(theta0+theta1+theta2);

 Jacobian(2,1)=l0*cos(theta0)+l1*cos(theta0+theta1)+l2*cos(theta0+theta1+t
heta2);
        Jacobian(2,2)=l1*cos(theta0+theta1)+l2*cos(theta0+theta1+theta2);
        Jacobian(2,3)=l2*cos(theta0+theta1+theta2);

       % Calculate the pseudo-inverse of the jacobian using 'pinv()':
        JPinv=pinv(Jacobian);
       % Update the values of the thetas by a small step:
         dx=x_e_target-x_e;
         dy=y_e_target-y_e;
         dtheta=0.1*JPinv*[dx;dy];
         theta0=theta0+dtheta(1);
         theta1=theta1+dtheta(2);
         theta2=theta2+dtheta(3);
        % Obtain end effector position x_e, y_e for the updated thetas:
        [x_1,y_1,x_2,y_2,x_e,y_e]=ForwardKinematics(l0,l1,l2, theta0,
theta1, theta2);

        %showing the end effector position for each iteration
           plot(x_e,y_e,"*r");
           xlim([-20 20]);
           ylim([-20 20])
           xticks(-20: 2: 20);
           yticks(-20: 2: 20);
           hold on
```

```
            grid on
            hold on
end

    % Draw the robot using drawRobot
    drawRobot(x_1,y_1,x_2,y_2,x_e,y_e);
```