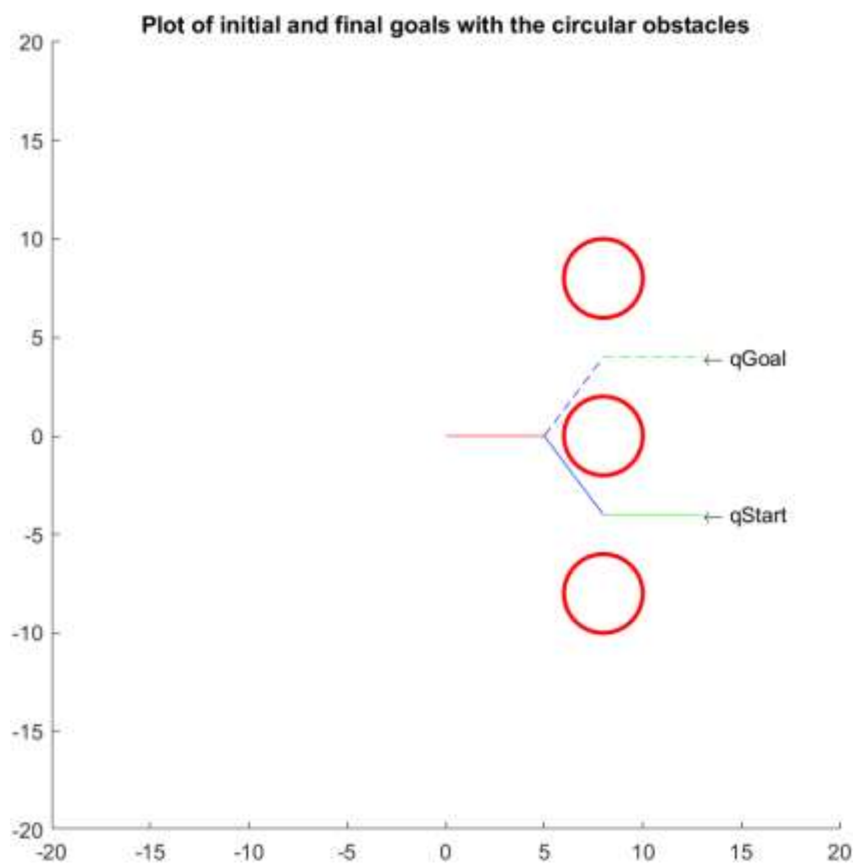
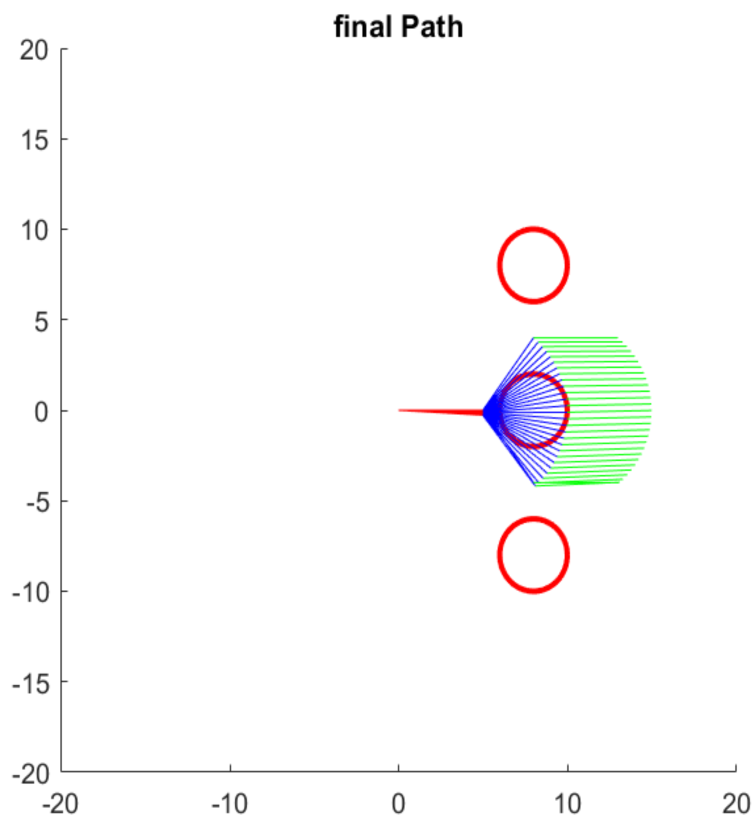
Project Assignment 2

1. The robot utilized is 3 degrees-of-freedom planar robotic manipulator, with three rotational joints
2. The initial and goal positions are shown as below.

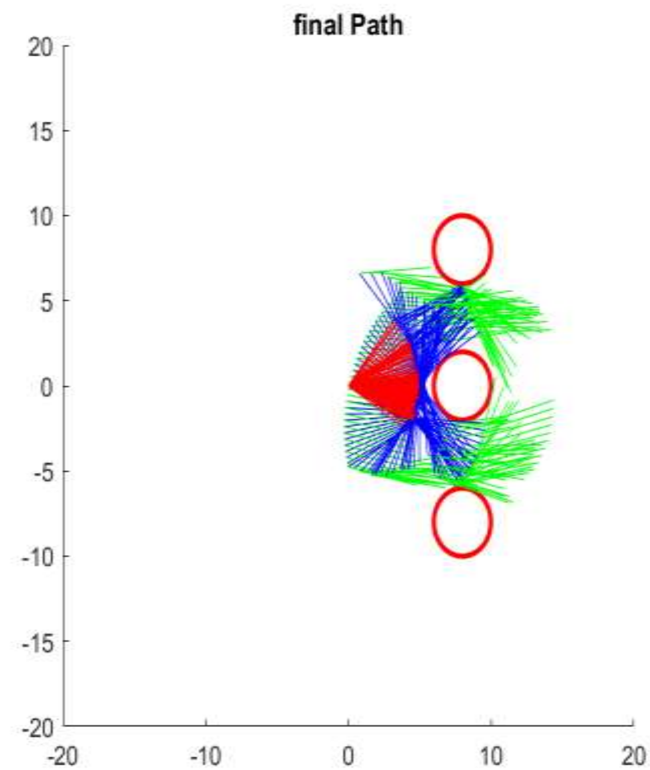


3. If we do not account for obstacles
   It takes very few nodes to find the path without obstacles



4. If we do account for the obstacles

5. It takes n = 1000, nodes to find the path with obstacles



I have use a third party library for building and accessing trees, the code of which is attached in the submission.
The coding involved going through 7 stages of development, which are described by the code in the appendix.

1. Code – part 1 – understanding the problem.
2. Code – part 2– developing the local planner.
3. Code – part 3– developing the collision checker.
4. Code – part 4- developing single_multiple function.
5. Code – part 5– developing extend_multiple function.
6. Code – part 6– developing rrt algorithm function.
7. Code – part 7–plotting the result.

Code – part 1 – understanding the problem:

```
% understanding the work space.

% There are three circular obstacles placed in the workspace of the robotic manipulator.

% The obstacles are centered at coordinates (8,0), (8,8), and (8,-8), and they all have radii of 2 units.

 % plotting qgoal and qstart


figure('Name','Obstacles')

title('Plot of initial and final goals with the circular obstacles')



 % Fix the axis limits.

    xlim([-20  20])

    ylim([-20  20])


    % Set the axis aspect ratio to 1:1.

axis square

viscircles([8,0],2);

hold on;

% Clear the axes.


viscircles([8,8],2);

hold on;


viscircles([8,-8],2);

hold on;


q_start =[0 -0.9273 0.9273];

q_goal = [0  0.9273  -0.9273];

a = 5;
```

```
x_initial = 0;
y_initial = 0;
theta1 = q_start(1,1);
theta2 = q_start(1,2);
theta3 = q_start(1,3);
x_final1 = a*cos(theta1);
y_final1 = a*sin(theta1);
x_final2 = a*cos(theta1) + a*cos(theta1 +theta2) ;
y_final2 = a*sin(theta1) + a*sin(theta1 +theta2) ;
x_final3 = a*cos(theta1) + a*cos(theta1 +theta2) + a*cos(theta1 +theta2 + theta3);
y_final3 = a*sin(theta1) + a*sin(theta1 +theta2) + a*sin(theta1 +theta2 + theta3);
line([x_initial,x_final1],[y_initial,y_final1],'Color','red');
hold on ;
line([x_final1,x_final2],[y_final1,y_final2],'Color','blue');
hold on ;
line([x_final2,x_final3],[y_final2,y_final3],'Color','green');
hold on ;
% line([x_initial,x_final3],[x_initial,y_final3],'Color','black');
hold on;
theta1 = q_goal(1,1);
theta2 = q_goal(1,2);
theta3 = q_goal(1,3);
x_final1g = a*cos(theta1);
y_final1g = a*sin(theta1);
x_final2g = a*cos(theta1) + a*cos(theta1 +theta2) ;
y_final2g = a*sin(theta1) + a*sin(theta1 +theta2) ;
x_final3g = a*cos(theta1) + a*cos(theta1 +theta2) + a*cos(theta1 +theta2 + theta3);
y_final3g = a*sin(theta1) + a*sin(theta1 +theta2) + a*sin(theta1 +theta2 + theta3);
line([x_initial,x_final1g],[y_initial,y_final1g],'Color','red','LineStyle','--');
```

```
hold on ;

line([x_final1g,x_final2g],[y_final1g,y_final2g],'Color','blue','LineStyle','--');

hold on ;

line([x_final2g,x_final3g],[y_final2g,y_final3g],'Color','green','LineStyle','--');

hold on ;


text(x_final3,y_final3,'\leftarrow qStart ')

text(x_final3g,y_final3g,'\leftarrow qGoal ')

% line([x_initial,x_final3],[x_initial,y_final3],'Color','black','LineStyle','--');
```

Code – part 2– developing the local planner:

%% local planner

```
function [success] = local_planner(q_near, q_int, step_size)

success = false;

disp('local_planner called ')

obstacle1 = [8, 0, 2];

obstacle2 = [8, 8, 2];

obstacle3 = [8, -8, 2];

% obstacles = [];

obstacles = [obstacle1; obstacle2 ; obstacle3];


 % Fix the axis limits.

 xlim([-16  16]);

 ylim([-16  16]);

 axis square

 viscircles([8,0],2);

 hold on;
```

```
 viscircles([8,8],2);
 hold on;

 viscircles([8,-8],2);
 hold on;

q2 = q_int;
q1 = q_near;
delta_q = q2 - q1;
delta_q = limitAngle(delta_q);
num_steps = ceil(norm(delta_q)/step_size); % be careful about angle correction
step = delta_q/ num_steps;
collision = false;
q = q1;
   for i = 1: num_steps %%4

       q = q + step ;
        collision = collision || check_collision( q, obstacles);
       if(collision == true)
           return;
       end
   end
success = not(collision);
end



function [doesTouch] = check_collision(q,obstacles)
```

```
disp('check_collision called ')
doesTouch = false;
a = 5;
x_initial = 0;
y_initial = 0;
theta1 = q(1,1);
theta2 = q(1,2);
theta3 = q(1,3);

x_final1 = a*cos(theta1);
y_final1 = a*sin(theta1);
x_final2 = a*cos(theta1) + a*cos(theta1 +theta2);
y_final2 = a*sin(theta1) + a*sin(theta1 +theta2);
x_final3 = a*cos(theta1) + a*cos(theta1 +theta2) + a*cos(theta1 +theta2 + theta3);
y_final3 = a*sin(theta1) + a*sin(theta1 +theta2) + a*sin(theta1 +theta2 + theta3);

line([x_initial,x_final1],[y_initial,y_final1],'Color','red');
hold on ;
line([x_final1,x_final2],[y_final1,y_final2],'Color','blue');
hold on ;
line([x_final2,x_final3],[y_final2,y_final3],'Color','green');
 hold on;
%    hold on ;
%    line([x_initial,x_final3],[x_initial,y_final3],'Color','black');

for i = 1 : length(obstacles)
    obstacles(i,:);
    collidesl1 = collisionCheck(obstacles(i,:),[x_final1,x_final2],[y_final1,y_final2] ); %2nd link
    collidesl2 = collisionCheck(obstacles(i,:),[x_final2,x_final3],[y_final2,y_final3] ); %3rd link
```

```
        doesTouch = false;
        doesTouch = collidesl1 || collidesl2;
        if(doesTouch == true)
            break;
        else
            continue;
        end
    end
end




function [Ta_mat] = limitAngle(Ta_mat)
        [m, n] = size(Ta_mat); %m rows, n columns
        for i = 1: m
            for j = 1: n
                if (Ta_mat(i,j) >= pi )
                    Ta_mat(i,j) = Ta_mat(i,j) - 2*pi;
                    elseif (Ta_mat(i,j) < - pi )
                    Ta_mat(i,j) = Ta_mat(i,j) + 2*pi ;
                end
            end
        end
    end
```

Code – part 3– developing the collision checker:

```
function [collides] = collisionCheck(obstacle,X,Y)
```

```
disp("collisionCheck called")

collides = false
% %% make a frame
%    xlim([0  16]);
%    ylim([0  16]);
%     axis square
% %% show a circle
%    viscircles([4,4],2);
% %% test points
%    x =[5,0,6,0,0,2];
%    y =[0,4,0,6,10,8];
%
% %% circle points
%    c1 = 4;
%    c2 = 4;
   C = [obstacle(1),obstacle(2)]
   r =  obstacle(3)
%% ALGORITHM
%    E is the starting point of the ray,
%    L is the end point of the ray,
%    C is the center of sphere you're testing against
%    r is the radius of that sphere
%    line([0,2],[10,8]);
   E1 = [X(1),Y(1)];
   L1 = [X(2),Y(2)];
   d = L1 - E1 %( Direction vector of ray, from start to end )
   f = E1 - C %( Vector from center sphere to ray start )
   a = dot( d,d )
```

```
    b = 2*dot( f,d ) ;
    c = dot( f,f ) - r*r ;
    discriminant = b*b-(4*a*c);
    if(discriminant < 0)
%        //no intersection
      collides = false;
    else
      discriminant = sqrt( discriminant );
      t1 = (-b - discriminant)/(2*a)
      t2 = (-b + discriminant)/(2*a)

      if( t1 >= 0 && t1 <= 1 )
%     // t1 is the intersection, and it's closer than t2
%     // (since t1 uses -b - discriminant)
%     // Impale, Poke
        collides = true;
      end
      if ( t2 >= 0 && t2 <= 1 )
        collides = true;
      end


    end




  end
```

Code – part 4– developing extend_single function:

```
function [result, Ta, q_target ] = rrt_extend_single_func(Ta, q_random, step_length)

    step_size = 0.01;

    disp("rrt_extend_single_func called")

    q_target= [];

    [q_near,indexnear] = find_nearest(Ta, q_random);

    q_int = limit( q_random, q_near, step_length); % be careful about angle correction


    result = local_planner(q_near, q_int, step_size);

    if (result == true)

        Ta = Ta.addnode( indexnear , q_int);

        q_target = q_int;

    end

end

%% find_nearest

function[q_near,indexnear] = find_nearest(Ta, q_random)

    disp("find_nearest called")

    n = nnodes(Ta); % number of nodes in Tree A

    Ta_mat = zeros(n,3);% magnitude matrix for Tree A initialised to zero

    diff_mat = zeros(1,n);

    for i = 1 : n

        Ta.get(i)

         diff_mat(1,i) = norm(limitAngle(Ta.get(i) - q_random))  % array of Tree A

    end

    index = find(diff_mat == min(diff_mat));

    q_near = Ta.get(index(1));

    indexnear = index(1);

end

%% limit
```

```
% The LIMIT function finds an intermediate configuration q_int from between q_near and q_target,
% such that q_int is at a distance of step_length from q_near.
% If the distance between q_near and q_target is less than step_length, it would return q_target
function [q_int] = limit( q_random, q_near, step_length)
disp("limit called")
delta_q = q_random - q_near ;
delta_q = limitAngle(delta_q);
step = step_length * delta_q / norm(delta_q) ;
% step_length = norm(step)
flag = step_length < norm(delta_q);
if(flag == true)
   q_int = q_near + step  ;
else
   q_int  = q_random;
end
end



%% random node generator
function[q_rand] = random_node_gen()
a = 0;
b = 2 * pi;
q_rand = a + (b-a).* rand(1,3);
% angleInDegrees = rad2deg(q_rand);
for i = 1: length(q_rand)
   if (q_rand(1,i) > pi)
      q_rand(1,i) = -( b - q_rand(1,i));
   end
end
```

```
end

%% limitAngle
function [Ta_mat] = limitAngle(Ta_mat)
    [m, n] = size(Ta_mat); %m rows, n columns
    for i = 1: m
      for j = 1: n
        if (Ta_mat(i,j) >= pi )
          Ta_mat(i,j) = Ta_mat(i,j) - 2*pi;
          elseif (Ta_mat(i,j) <- pi )
          Ta_mat(i,j) = Ta_mat(i,j) + 2*pi ;
        end
      end
    end
  end
```

Code – part 5– developing extend_multiple function:

```
function [result, Tb, q_connect] = rrt_extend_multiple_func(Tb,q_target,step_length)

step_size = 0.1;

disp("rrt_extend_multiple_func called")
q_connect= [];
result = false;
[q_near,indexnear]= find_nearest(Tb, q_target);
q_int = limit( q_target, q_near, step_length); % be careful about angle correction
```

```
q_last= q_near;
num_steps= ceil(norm(limitAngle(q_target-q_near))/step_length);

for i=1:num_steps
   result = local_planner(q_int, q_last, step_size);

   if (result == true)
      %          [ t node1 ] = t.addnode(1, 'Node 1'); %% attach to root

      [Tb,  idx]= Tb.addnode(indexnear , q_int);
      q_connect = q_int;
      if (i< num_steps)
         q_last = q_int;
         q_int = limit( q_target, q_int, step_length);
         indexnear = idx; %%% have added qint, its index in that tree
      end
   else
      return;
   end
end

end

%% find_nearest
function[q_near,indexnear] = find_nearest(Ta, q_random)
n = nnodes(Ta); % number of nodes in Tree A
   Ta_mat = zeros(n,3);% magnitude matrix for Tree A initialised to zero
   diff_mat = zeros(1,n);
```

```
    for i = 1 : n
        Ta.get(i)
        diff_mat(1,i) = norm(limitAngle(Ta.get(i) - q_random))  % array of Tree A
    end

    min(diff_mat);
    index = find(diff_mat == min(diff_mat));
    q_near = Ta.get(index(1));
    indexnear = index(1);


end

%% limit
% The LIMIT function finds an intermediate configuration q_int from between q_near and q_target,
% such that q_int is at a distance of step_length from q_near.
% If the distance between q_near and q_target is less than step_length, it would return q_target
function [q_int] = limit( q_random, q_near, step_length)
disp("limit called")
delta_q = q_random - q_near;
delta_q = limitAngle(delta_q);
step = step_length * delta_q / norm(delta_q) ;
flag = step_length < norm(delta_q);
if(flag == true)
    q_int  = q_near + step;
else
    q_int  = q_random;
end
```

```
end

%% limit Angle

  function [Ta_mat] = limitAngle(Ta_mat)

    [m, n] = size(Ta_mat); %m rows, n columns

    for i = 1: m

      for j = 1: n

        if (Ta_mat(i,j) >= pi )

          Ta_mat(i,j) = Ta_mat(i,j) - 2*pi;

          elseif (Ta_mat(i,j) < - pi )

          Ta_mat(i,j) = Ta_mat(i,j) + 2*pi ;

        end

      end

    end

  end
```

Code – part 6– developing rrt algorithm function:

```
function [] = main()


clc;

%rng(1);

disp("main called")

q_start =[0, -0.9273 ,0.9273];

q_goal = [0 , 0.9273 , -0.9273];

Ta = tree(q_start);

Tb = tree(q_goal);

success  = false;

max_nodes = 1000;

step_length = 0.1;

% step_length = 1;
```

```
q_connect=[];

result2 = false;

result = false;



for i = 1: max_nodes

   q_rand = random_node_gen(); %% find the next point to extend to

   [result, Ta, q_target] = rrt_extend_single_func(Ta, q_rand, step_length); % extend from Ta to this new point using this step length

   if(result == true)

      disp("1st True")

      [result2, Tb, q_connect] = rrt_extend_multiple_func(Tb, q_target,step_length);

      if (result2 == true) % connected the two trees

         disp("2nd True")

         success = true;

         break;

      end

   end

   [Ta,Tb] = swap(Ta, Tb);

end

plotFinal(success, q_connect, Ta, Tb);

end


%% random_config()

function[q_rand] = random_node_gen()

disp("random_node_gen called");

a = 0;

b = 2 * pi;

q_rand = a + ( b-a ).* rand(1,3);
```

```
for i = 1: length(q_rand)
    if (q_rand(1,i) >= pi) % if value is greater than 180, we get a negative degree
        q_rand(1,i) = -( b - q_rand(1,i));
    end
end
end
%% swap
function [Ta,Tb] = swap(Ta, Tb)
temp = Ta;
Ta = Tb ;
Tb = temp ;
disp("swapped");
end
```

Code – part 7– plotting the result

```
function [] = plotFinal(success, q_connect, Ta, Tb)
disp("plotFinal called");
nodesA = nnodes(Ta);
nodesB = nnodes(Tb);
pathFromA = zeros(nodesA ,3);
pathFromB = zeros(nodesB ,3);
if success == true
    for nA = nodesA :-1:1 %% tracing back each parent from final node
        elem =  Ta.getparent(nA);
        if(elem == 0)
            parent = -1 ;
        else
        parent = Ta.get(elem);
        end
        pathFromA(nodesA - nA + 1,:) = parent;
```

```
    end

    for nB = nodesB :-1:1
        elem =  Tb.getparent(nB);
        if(elem == 0)
            parent = -1;
        else
        parent = Tb.get(elem) ;
        end
        pathFromB(nodesB - nB + 1,: )= parent;
    end
    pathFromA
    pathFromB
    pathFromA(nodesA,:) = []; % remove the last element i.e. [-1,-1,-1]
    pathFromB(nodesB,:) = [];
    path = [pathFromB; q_connect; pathFromA] % make the whole path
    plotThePath(path);
else
    disp("no plot");
    for i=1 : nodesA
        pathFromA(i,:) = Ta.get(i);
    end
    plotThePath(pathFromA)
    for i=1 : nodesB
        Tb.get(i);
        pathFromB(i,:) = Tb.get(i);
    end
    plotThePath(pathFromB)
end
```

```
end
%%
function [] = plotThePath(path)
% path
% length(path)
disp("plotThePath called");

figure();
title('final Path');



% Fix the axis limits.
   xlim([-20  20]);
   ylim([-20  20]);

% Set the axis aspect ratio to 1:1.
axis square
viscircles([8,0],2);
hold on;
% Clear the axes.

viscircles([8,8],2);
hold on;

viscircles([8,-8],2);
hold on;
   a = 5;
   x_initial = 0;
```

```
    y_initial = 0;
    for q = 1 : size(path,1)
        q;
        theta1 = path(q,1);
        theta2 = path(q,2);
        theta3 = path(q,3);

        x_final1 = a*cos(theta1);
        y_final1 = a*sin(theta1);
        x_final2 = a*cos(theta1) + a*cos(theta1 +theta2);
        y_final2 = a*sin(theta1) + a*sin(theta1 +theta2);
        x_final3 = a*cos(theta1) + a*cos(theta1 +theta2) + a*cos(theta1 +theta2 + theta3);
        y_final3 = a*sin(theta1) + a*sin(theta1 +theta2) + a*sin(theta1 +theta2 + theta3);

        line([x_initial,x_final1],[y_initial,y_final1],'Color',"red");
        hold on ;
        line([x_final1,x_final2],[y_final1,y_final2],'Color',"blue");
        hold on ;
        line([x_final2,x_final3],[y_final2,y_final3],'Color',"green");

        hold on;
    end

end
```