Implementation of the Rapidly-exploring Random Tree and Potential Field Motion Planning Algorithms in the UR5 Manipulator

A Mini Project for the Summer Course
Robots in Context

Group #1

Heinz Alfred Juan Tong
Josep Quintana I Torres
Mathias Christian Clausen
Shuoshi Li
Sebastião Rocha

15 August 2019

Introduction

Robots, as defined by the dictionary, are "a machine that resembles a living creature in being capable of moving independently (as by walking or rolling on wheels) and performing complex actions (such as grasping and moving objects)" (Robot, n.d.). How does it move or do complex tasks independently though? These are done with the use of different kinds of algorithms programming the robots to become independent. For the robot to move from one point to another while avoiding collision with obstacles positioned in the free space, it´s required to implement motion planning algorithms.

For this mini project, we implemented two types of motion planning algorithms at different times to observe how each of them works and to compare the two. These algorithms are known as the Rapidly-exploring Random Tree (RRT) and the Potential Field (PF) which would be further discussed below. These algorithms will be implemented in the Universal Robots 5 manipulator (UR5). Programming of the robot will be done using MATLAB and URscripts.

Project Objectives

- To program the UR5 to perform motion planning using the RRT algorithm
- To program the UR5 to perform motion planning using the PF algorithm.
- To discuss how each of the two motion planning algorithms were implemented.
- To compare the two types of motion planning algorithms that were implemented.
- To discuss the problems encountered, and in the case of implementation failure, discuss the problems and recommend possible alternative ways to implement the algorithms.

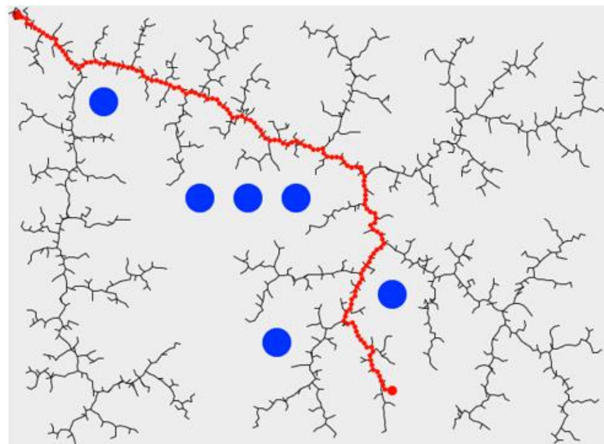
Theoretical Background

Rapidly-exploring Random Tree (RRT)

The RRT is a probabilistic single-query planning algorithm where it builds a single tree from the initial configuration to the goal configuration. The algorithm is probabilistic because random points in the robot's configuration space are generated, and afterwards it´s checked if the point is in collision with an obstacle, or if the robot will undergo a collision if it moves to that random configuration in a straight line (direction). It´s a single-query type because it is based on one thread only rather than allowing the road mapping (creation of random points in space) phase and path

creation phase (finding a path from start to goal) run without strict sequence (Alandes, 2015). RRT works if the all obstacles are known.

RRT works by generating a random point within the configuration space for each main iteration. If the random point is within the boundaries of the known obstacles, the random point is discarded. Otherwise, new extensions towards the random point are continuously created at a defined step-size from part of the tree nearest to the random point until it hits an obstacle or the random point. Afterwards, it generates another random point and follows such process repeatedly until the tree (robot) reaches the goal configuration. This would eventually lead to it being a probabilistically complete algorithm (Choset, Lynch & Hutchinson, 2004).



Rapidly-exploring Random Tree, (Akkaya, n.d.)

Due to the huge number of configuration space points that are generated to move the robot from its initial configuration to its goal configuration, stuttering would be evident during the real-world application. In order to avoid such phenomenon, we need to smoothen or shorten the path that the robot takes from initial to goal; this called postprocessing. The usual postprocessing algorithm involves connecting the two neighbors of a point in the tree and checking if there will be a collision. If a collision is observed, the tree is left as is; if there is no collision, the neighbors of the point will be connected to each other and the point is deleted. Thereby removing one configuration point in the path and reducing the stutter (Choset, Lynch & Hutchinson, 2004).

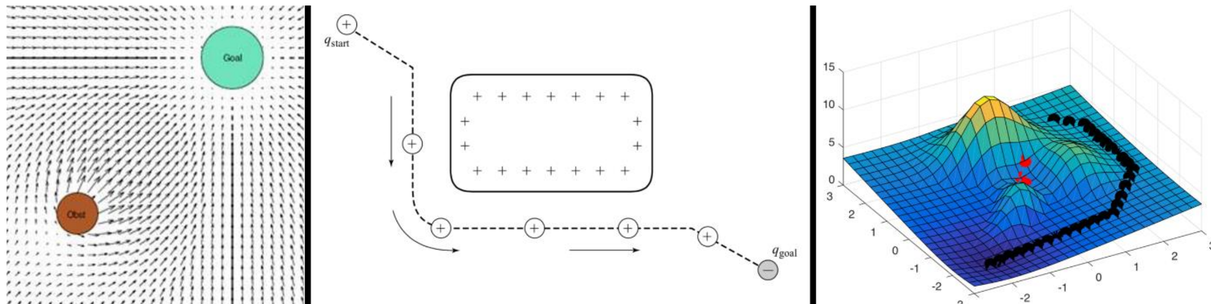
Potential Field (PT)

Potential field method is a common method used for local path planning. This method assumes that the robot moves in a virtual force field.

Potential field is a resultant force field, which consists of the attractive component that guides the robot to reach q_{goal} and the repulsive component generated by the obstacles that repels the robot

away from the boundary of *QO*. The resultant force at each point in the path equals the sum of all repulsive forces and attractive forces at that point.

$$U(q)=U_{att}(q)+U_{rep}(q)$$



(Path Planning—Potential Field Method)

The figures above, show the potential field method. The first figure illustrates the robot as being a positive particle moving towards a negative charged goal. The obstacle is positive, so the robot will be repelled away from the obstacle.

In the second figure, it´s possible to observe a workspace where the robot must reach the target destination (attraction), there´s also an obstacle what will repel the robot avoiding the collision.

The third figure shows the initial position of the robot on a higher ground in the upper right corner. The target position to be reached is under the "foot of the hill", which forms a potential field. Under the guidance of this potential field, the robot is now capable of avoiding the obstacles (peaks) and reach the target point with success.

If the attractive and repulsion are equal, the algorithm may fall into the problem of local minimum, where the algorithm thinks that it has found the goal. To avoid this situation a random perturbation (force) can be added to make the robot jump out of the local minimum state, and then avoiding excessive or even infinite path solutions. This is the reason why sometimes the algorithm is not able to complete the task.

For path planning, an important concept is that it should provide a speed and direction control command to the vehicle so that the underlying controller can follow this input to perform motion control operations. The robot is regarded as a point in the configuration space under the influence of the potential field *U*. The structure of potential *U* is as follows: the robot can be attracted to the final position q_{goal}, while being excluded from the boundary of the obstacle area *QO*. If possible, *U* should be constructed to make there is only a single global minimum and no local minimum in the potential field UU. The planning process takes place incrementally; at each configuration point *q*, the artificial force generated by the potential field is defined as the negative gradient of the potential field [-U (q)], indicating the most likely direction of local motion.

For the attractive potential field, it should satisfy several requirements;

1. $Uatt, Urep$ should be the monotonic incremental function of the distance from Oi to the target location, like Conic Well Potential:

$$U_{att,i}(q)=\|O_i(q)-O_i(q_f)\|$$

2. The attractive force on Oi in the workspace is equal to the negative gradient of $Uatt$, $iUatt$:,

$$F_{att,i}=-\nabla U_{att,i}(q)=-\xi_i(o_i(q)-o_i(q_f)).$$

For the repulsive potential field, it should make the robot repel and stay away from obstacles, and when the robot is far away from obstacles, obstacles should have little or no effect on the movement of the robot.

After constructing potential fields in the workspace, we need to map the workspace force to joint torque, thus the artificial force is induced at the origin of the DH coordinate system Oi of the robot-arm.

Jv includes the first three rows of Jacobian matrix of the manipulator. We don't use the last three lines for we only consider gravity and repulsion in the workspace, without attraction and repulsion moments in the workspace. For each Oi, a suitable Jacobian matrix must be constructed.

Comparison Between the Two Algorithms

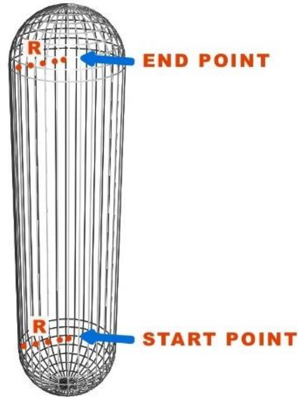
The algorithms work and perform using different approaches. The RRT algorithm follows a probabilistic method, by creating random points across the configuration space. The algorithm has a slow convergence to the optimal path solution for the problem. This results in high memory usage and time required for achieving the best path solution.

The Potential Field algorithm uses a gravitational force field, composed by repulsive and attractive forces, to guide the robot over the workspace in a characteristic method, this may result in poor performance in narrow passages, prone to get stuck in local minima situations and problems related to symmetrical obstacles.

Methodology

Obstacle Generation and Placement

In order to generate multiple obstacles in the simulator, that the robot should avoid, we used a Nx7 matrix called *obs* where n is the index of the nth-obstacle. The obstacles are generated with the geometrics like a capsule as shown below which is defined by the 7 values. The 7 columns represent the positions of the start and end points of the capsule followed by the capsule radius.

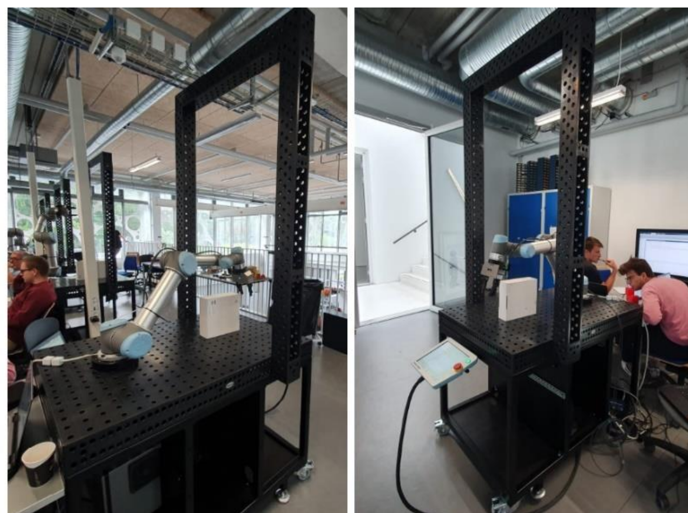


Obstacle defined by [x_1 y_1 z_1 x_2 y_2 z_2 r]

Capsules are used to model the obstacles because you could make almost any obstacle by connecting many capsules of different lengths and radius. Take a box for example, by positioning multiple capsules one after the other, you would end up with cube like object with rounded edges. When generating obstacles for the simulator, we want to make the obstacles a bit bigger than they are. That way, there will be allowable tolerances: in case of accidental collision in the model object's border, it will not necessarily damage the actual robot.

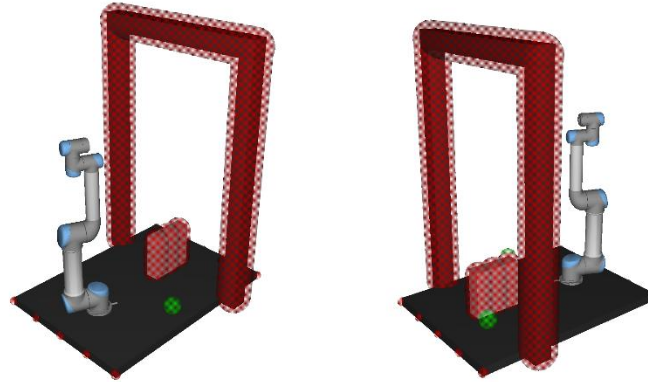
The Robot Workspace

The robot workspace in real life is shown in the figure below. It consists of the robot itself; the table it is placed on, the pillars of the frame, and finally, the obstacle in the middle of the path.



Real Life Robot Workspace

Using the capsules to generate objects, we modelled the real-life robot workspace. Below is a figure showing a visual effect of what the algorithm would perceive from the nx7 *obs* matrix. A total of 9 capsules were used to model this virtual robot workspace.



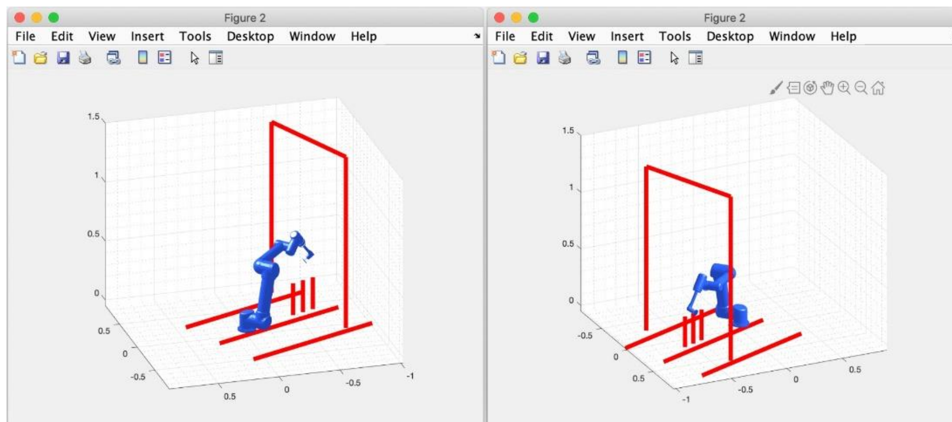
Virtual Robot Workspace

Since the robot was mounted on the table at an angle of 22.5°, we rotated all the obstacles as well as the initial and final configurations about the z-axis using the rotation matrix R_z

$$R_z = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$

Where: $\theta = 22.5°$

The final result inside Matlab UR simulator is shown on the figure below.



Virtual Robot Workspace inside Matlab

Rapidly-exploring Random Tree

The RRT motion planning algorithm was implemented as an extension of the 2-dimensional version of the algorithm. The goal was to extend it from 2 dimensions to 6 dimensions. This output of the algorithm was an ix6 matrix where i was the number of configurations the robot moved to and the 6 columns was the angles for each of the UR5's degrees of freedom in radians. The angles of the initial and goal configurations were found by moving the UR5, and noting the values on the teach pendant. To input the configurations in the simulation and the robot, we had to convert it from degrees to radians first via the function *degtorad( )*.

For the implementation of the RRT, two main scripts were created. These are the MPExtendRRT.m and the MPExtendTree.m; where MPExtendTree.m is an auxiliary function for the MPExtendRRT.m function.

MPExtendRRT.m

The MPExtendRRT.m script has 5 main processes. First it finds a point within the configuration space of the UR5; next it checks which part of the tree is nearest to the new point; then it calls the MPExtendRRT function; performs post processing after finding a path to the goal; and finally, calls OutputMovesForUR.m function.

Below is a loop which runs until either the maximum number of iterations is reached, or the goal is reached by the robot. The following if and else statements is the probabilistic part of the function. The rand() function creates a random number between 0 and 1.This means that there is a 30% chance that the point is generated with the goal configuration. Likewise, this means that there is a 70% that the point is generated randomly by the function *SampleState(). This i*s a support code to generate random points within the configuration space.

```
    while mp.vidAtGoal <= 0 && iter < params.maxiteration
        % Implement the extension of the RRT algorithm inside while loop
here ...
        if rand() < 0.3
            %try goal
            sto = params.goal;
        else
            %generate random point instead
            sto = SampleState();
        end
```

Below is a for loop that checks the distance between the randomly generated point and each part of the tree. The index of the point in the tree with the shortest distance is saved as *dmin* until all points in the tree is checked. In this implementation the points are called nodes.

```
n       = size(mp.nodes,1);
for k = 1:n
    %calculate distance
    d = norm(sto - mp.nodes(k, :));

    if d < dmin
        %find a good node
        dmin = d;
        vid  = k;
    end
end
```

Next the MPExtendTree function is called with the current generated point *sto* and the index of this point *vid*. Further details of this function will be explained in the later part of this section.

```
MPExtendTree(vid, sto);
```

The part of the script shown below is implemented after a path to the goal is found. The post processing function *SmoothPath()* is a support code used to reduce the number of configurations the robot undergoes to reach the goal configuration. After post processing, it calls the *OutputMovesForUR()* function to output the configurations of the motion to the UR5 in the right format. *OutputMovesForUR()* will be further discussed in another section of this paper.

```
        if mp.vidAtGoal >= 1
            JointTrajectory  = MPGetPath();
            %smooth the generated path
            JointTrajectory_smooth = SmoothPath(JointTrajectory);
            %output the output moves
            OutputMovesForUR(JointTrajectory_smooth);
            %display the final result
            Draw(JointTrajectory_smooth);
        end
    end
```

MPExtendTree.m

The MPExtendTree.m script extends the existing tree from the nearest point towards the random point generated, with a maximum of 10 steps. While doing that, it checks if it collides with an obstacle. If it does not collide, it records the new point in the tree, keeps track of the new point's parent, i.e. its predecessor, increases the number of the parent's children and initialize the number of the new point's children to 0. If the goal configuration is reached with the next point or 10 steps were already made, it returns to the MPExtendRRT.m script.

The part of the script shown below is where we save values to be used later. The *dstep* saves the step size from one part of the tree to another. *C_curr* is the vector from the nearest point in the

tree to the random point generated. The u is the normalized form of *C_curr* and scaled to the step size. We also limited the number of steps the robot could move in one direction before another random point is generated.

```
% You can make use of the global variables params and mp to access the
% necessary information

% Add your code here ...

dstep     = params.distOneStep;
C_curr    = sto - mp.nodes(vid, :);
d         = norm(C_curr);
u         = dstep * C_curr / d;
nrSteps = 10; %ceil(d / dstep);
```

The part of the script shown below is how the *MPExtendTree()* function extends the tree. It increments the current position in the tree by u and then checks if there are collisions or not. In case of collision, the function returns back to the MPExtendRRT.m script.

```
for k = 1:nrSteps
    params.robot = mp.nodes(vid, :) + u;
    if IsValidState() == 0
        %UR robot configuration is not valid
        return;
    end
```

The part of the script shown below includes storing indexing information of the point and its neighbors as well as checking if the goal configuration has been met. The new node of the tree is stored in the in the *mp.nodes* matrix; its parent node is recorded; the number of children its parent node increases by one because of the node itself; and finally, the node itself now has an entry in the *mp.nchildren* array describing that it has 0 children. If the robot actually did reach the goal, its index is saved in the *mp.vidAtGoal* for future uses.

```
        %save this new node
        n                      = size(mp.nodes,1);
        mp.nchildren(vid)      = mp.nchildren(vid) + 1;
        mp.nodes(n + 1, :)     = params.robot;
        mp.parents(n + 1)      = vid;
        mp.nchildren(n + 1)    = 0;

        if HasRobotReachedGoal() == 1
            %robot has reached goal
            mp.vidAtGoal = n + 1;
            return;
        end
        vid = n + 1;
    end
end
```

Potential Field

The angles of the initial and goal configurations as taken from the UR5's teach pendant is displayed in degrees. In order for us to input these configurations, we had to convert it to radians first via MATLAB using the function *degtorad( )*.

The part of the script file *PotentialField.m* initializes the variables that we will be dealing with throughout the algorithm. The *DHTransformation()* function calculates the transformation matrix for each of the 6 joint angles of the goal configuration. Then we substitute the symbolic variables and evaluate the expression to get scalars values of x y z.

```
MPInitialize(C_ini);
ParaInitialize(C_ini, C_goal, Obs);

syms Q1 Q2 Q3 Q4 Q5 Q6
p = [0 0 0.15 1]';

iter      = 1;
dstep     = params.distOneStep;
vid       = 1;
nrLinks   = 6;
dhgoal = DHTransformation(C_goal, 6);
G = subs(dhgoal, [Q1 Q2 Q3 Q4 Q5 Q6], params.goal);
G = eval(G)*p;
G = G(1:3,1);
```

A while loop is implemented so the algorithm runs until it either reaches the goal configuration or reaches the maximum number of iterations. The for loop below, was originally meant to calculate the repulsive force affecting each control point.

```
while mp.vidAtGoal <= 0 && iter <= params.maxiteration
    %for i = 1:nrLinks
      %  for j = 1:i
              %--To check repulsive force for each joint
      %  end
    %end
```

The part shown below is used to retrieve the workspace coordinate of the current configuration's end effector or the TCP.

```
dhvid  = DHTransformation(mp.nodes(vid, :), 6);
pp = dhvid*p;
pp = pp(1:3);
```

The part shown below calculates the Jacobian of the current TCP coordinate in the workspace using *diff()* function to do partial derivatives. It then takes the unit vector from current position to

goal position and multiplies it by the transpose of the Jacobian. It is then scaled down and used to perform minute movements towards the goal.

```
        J = [diff(pp(1), Q1) diff(pp(1), Q2) diff(pp(1), Q3) diff(pp(1), Q4)
diff(pp(1), Q5) diff(pp(1), Q6);
             diff(pp(2), Q1) diff(pp(2), Q2) diff(pp(2), Q3) diff(pp(2), Q4)
diff(pp(2), Q5) diff(pp(2), Q6);
             diff(pp(3), Q1) diff(pp(3), Q2) diff(pp(3), Q3) diff(pp(3), Q4)
diff(pp(3), Q5) diff(pp(3), Q6)];
        J = double(subs(J,[Q1 Q2 Q3 Q4 Q5 Q6], mp.nodes(vid, :)));

        Psub = subs(pp, [Q1 Q2 Q3 Q4 Q5 Q6], mp.nodes(vid, :));
        d = G - eval(Psub);
        d = d/norm(d);
        u = J'*d;
        u = u*dstep;
        params.robot = mp.nodes(vid, :) + u'; % New configuration
```

The part shown below stores the current configuration node that will be used in the *MPGetPath( )* function for simulation. It then checks if the robot has reached the goal and updates a variable mp.vidAtGoal to exit the loop in the next iteration.

```
n                       = size(mp.nodes,1);
mp.nchildren(vid)       = mp.nchildren(vid) + 1;
mp.nodes(n + 1, :)      = params.robot;
mp.parents(n + 1)       = vid;
mp.nchildren(n + 1)     = 0;

if HasRobotReachedGoal() == 1
    mp.vidAtGoal = n + 1;
    return;
end

iter = iter + 1;
vid = vid + 1;
```

Sending the Path to UR5

To instruct the UR5 motion required from initial configuration to goal configuration, we created a generalized function that will be used for both motion planning algorithms (RRT and Potential Field), the OutputMovesForUR.m function file. This function will receive the generated path calculated by the respective motion planning algorithm, the path consists in a ix6 matrix where i is the number of configurations the robot moves (frames) to and the 6 columns are the angles for each of the UR5's degrees of freedom in radians.

For each robot move (frame configuration) the function will output the UR command (*movej*) with the respective rotation values for each joint, plus the acceleration, velocity, time and blend radius. The robot will start from the initial configuration and move to the goal configuration, wait 2 seconds on the goal and then go back to the initial configuration, waiting there 2 seconds.

Results

Rapidly-exploring Random Tree (RRT)

The RRT motion planning algorithm was successfully implemented with the RRT scripts discussed above. The UR5 managed to move from its initial configuration to its goal configuration, without any collision.

We noticed that for every obstacle/capsule we generated within the robot's workspace, the number of iterations it took to find a path increased even though it was not even in the way of the path.

At the first we were unable to make the UR5 move from its position. It took quite a bit of time to find the problem, but we later found out that either our initial or goal configuration was beyond the joint limits of the UR5, as were not able to see this while viewing the simulation.

We also had some problems with the orientation of everything inside the workspace since the robot had a 22.5 degree offset. We initially tried to rotate the robot's base, where we believed the offset was, by -22.5 degrees for them to cancel each other but that lead us to encounter some problems with the joint limits sometime later. We eventually decided that we just needed to rotate everything (the entire map, composed by the obstacles and the initial and goal positions) beside the UR5 by 22.5 degrees about the robot's base axis. This solution worked for us.

There are still some cases where the robot stutters a bit in its generated motion. This is because the motion generated by the algorithm is still a bit complex after the post processing. If you were to count the number of stops it makes throughout the motion, you could determine the number of rows in the c x 6 motion matrix that is generated by the algorithm to provide the UR5 its motion.

Potential Field

The goal was to implement the PF algorithm in the mini project using the support codes for the RRT method. The program was supposed to be an extension of the 2D potential field algorithm.

Our initial approach to the problem was to find a way to calculate only the attractive potential from the end-effector to the goal. If we succeeded in making the robot move from the initial configuration to the goal configuration, then afterwards the repulsive forces could be added.

We struggled a lot with the script, and in the end the algorithm could not move the robot to the goal, but ended up halfway, where it oscillated until the maximum number of iterations was reached.

We tried calling the function with the initial configuration and the goal configuration being the same values, but the simulator didn't appear, but instead just outputted an empty matrix.

We wrote the configuration of the robot out to the command window, to see if the robot moved from its initial configuration to its goal configuration. The configurations showed that the robot was moving in the right direction.

Reflection

After learning and getting contact with several algorithms used for motion planning for robots during the first week for this summer course, the students were assigned with a mini project related to the subject, especially with the RRT and the Potential Field algorithms. The group tried to accomplish and implement both algorithms, the RRT algorithm was implemented with success and works quite well, generating a path for the UR robot avoiding all the obstacles inside the workspace. The group then tried to implement the Potential Field algorithm, but unfortunately there are still problems related to the generated path for the robot. However, we believe that with more time and energy it could eventually be concluded as the majority of the algorithm is already implemented.

As a final reflection, we would like to recall that the group had no previous experience with UR robots, and many of us have no background nor experience in the robotics field. Still, the group managed to understand the main features present in the motion planning algorithms and get knowledge in this robotics field.

Appendix

Rapidly-exploring Random Tree (RRT)

MPExtendRRT.m

```
function [JointTrajectory, JointTrajectory_smooth] = MPExtendRRT(C_ini,
C_goal, Obs)
% C_ini is the initial configuration of UR5 (dimensino: 1*6, unit: radian)
% C_goal is the goal configuration of UR5 (dimensino: 1*6, unit: radian)
% Obs represents all the capsule obstacles in the workspace (dimension: n*7)
% format of each row of Obs: [P_ini,  P_end,  r] where P_ini is the x-, y-
% and z- coordinates of the initial point of the interal line segment of
% the capsule (diemsion: 1*3 unit: meter), P_end is the x-, y-
% and z- coordinates of the end point of the interal line segment of
% the capsule (diemsion: 1*3 unit: meter), r is the radius of the capsule
% (scalar, unit: meter)

% please define all these three inputs before runing this main function

    global mp;
    global params;

    JointTrajectory = [];
    JointTrajectory_smooth = [];
    % JointTrajectory is the original solution path and
JointTrajectory_smooth
    % is the path after post-processing the path JointTrajectory

    % JointTrajectory_smooth is the joint trajectories of all the six joints
of UR5,
    % which you should transform them into the format of UR script and deploy
them on the real robot.

    % JointTrajectory_smooth is a matrix whose dimension is n * 6 (unit:
radian)

    MPInitialize(C_ini);
    ParaInitialize(C_ini, C_goal, Obs);

    iter = 0;
    dmin    = Inf;
    vid     = -1;

    while mp.vidAtGoal <= 0 && iter < params.maxiteration
        % Implement the extension of the RRT algorithm inside while loop
here ...
        if rand() < 0.3
            %try goal
            sto = params.goal;
        else
            %generate random point instead
```

```
            sto = SampleState();
        end

        n      = size(mp.nodes,1);
        for k = 1:n
            %calculate distance
            d = norm(sto - mp.nodes(k, :));

            if d < dmin
                %find a good node
                dmin = d;
                vid  = k;
            end
        end
        MPExtendTree(vid, sto);
        %increase the iteration
        iter = iter + 1;
        %display iteration
        if mod(iter, 50) == 0
            fprintf('Iteration = %g\n', iter);
        end
    end

    if mp.vidAtGoal >= 1
        JointTrajectory  = MPGetPath();
        %smooth the generated path
        JointTrajectory_smooth = SmoothPath(JointTrajectory);
        %output the output moves
        OutputMovesForUR(JointTrajectory_smooth);
        %display the final result
        Draw(JointTrajectory_smooth);
    end
end
```

MPExtendTree.m

```
function [] = MPExtendTree(vid, sto)
% Extend the tree from the state with index vid toward the state sto
% At each time, make a small step with magnitude params.distOneStep
% Add each intermediate valid state to the tree data structure.
% Stop as soon as an intermediate invalid state is encountered
%   - You can check state validity by first setting the robot position and
%   - then calling the function IsStateValid
% Also stop if an intermediate state reaches the goal
%   - You can check if robot has reached the goal by first setting the
%   - robot position and then calling the function HasRobotReachedGoal

    global mp;
    global params;

    mp.sto = sto;
    mp.vidNear = mp.nodes(vid,:);

    % You can make use of the global variables params and mp to access the
```

```
    % necessary information

    % Add your code here ...

    dstep     = params.distOneStep;
    C_curr    = sto - mp.nodes(vid, :);
    d         = norm(C_curr);
    u         = dstep * C_curr / d;
    nrSteps = 10; %ceil(d / dstep);

    for k = 1:nrSteps
        params.robot = mp.nodes(vid, :) + u;
        if IsValidState() == 0
            %UR robot configuration is not valid
            return;
        end

        %save this new node
        n                     = size(mp.nodes,1);
        mp.nchildren(vid)     = mp.nchildren(vid) + 1;
        mp.nodes(n + 1, :)    = params.robot;
        mp.parents(n + 1)     = vid;
        mp.nchildren(n + 1)   = 0;

        if HasRobotReachedGoal() == 1
            %robot has reached goal
            mp.vidAtGoal = n + 1;
            return;
        end
        vid = n + 1;
    end
end
```

Potential Field

PotentialField.m

```
function [JointTrajectory, JointTrajectory_smooth] = PotentialField(C_ini,
C_goal, Obs)
% C_ini is the initial configuration of UR5 (dimensino: 1*6, unit: radian)
% C_goal is the goal configuration of UR5 (dimensino: 1*6, unit: radian)
% Obs represents all the capsule obstacles in the workspace (dimension: n*7)
% format of each row of Obs: [P_ini,  P_end,  r] where P_ini is the x-, y-
% and z- coordinates of the initial point of the interal line segment of
% the capsule (diemsion: 1*3 unit: meter), P_end is the x-, y-
% and z- coordinates of the end point of the interal line segment of
% the capsule (diemsion: 1*3 unit: meter), r is the radius of the capsule
% (scalar, unit: meter)

    % please define all these three inputs before runing this main function
    global mp;
    global params;
```

```
    JointTrajectory = [];
    JointTrajectory_smooth = [];
    % JointTrajectory is the original solution path and
JointTrajectory_smooth
    % is the path after post-processing the path JointTrajectory

    % JointTrajectory_smooth is the joint trajectories of all the six joints
of UR5,
    % which you should transform them into the format of UR script and deploy
them on the real robot.

    % JointTrajectory_smooth is a matrix whose dimension is n * 6 (unit:
radian)

    MPInitialize(C_ini);
    ParaInitialize(C_ini, C_goal, Obs);

    syms Q1 Q2 Q3 Q4 Q5 Q6
    p = [0 0 0.15 1]';

    iter      = 1;
    dstep     = params.distOneStep;
    vid       = 1;
    nrLinks   = 6;
    dhgoal = DHTransformation(C_goal, 6);
    G = subs(dhgoal, [Q1 Q2 Q3 Q4 Q5 Q6], params.goal);
    G = eval(G)*p;
    G = G(1:3,1);

    while mp.vidAtGoal <= 0 && iter <= params.maxiteration
        %for i = 1:nrLinks
          %  for j = 1:i
                    %--To check repulsive force for each joint
          %  end
        %end

        dhvid  = DHTransformation(mp.nodes(vid, :), 6);
        pp = dhvid*p;
        pp = pp(1:3);

        J = [diff(pp(1), Q1) diff(pp(1), Q2) diff(pp(1), Q3) diff(pp(1), Q4)
diff(pp(1), Q5) diff(pp(1), Q6);
             diff(pp(2), Q1) diff(pp(2), Q2) diff(pp(2), Q3) diff(pp(2), Q4)
diff(pp(2), Q5) diff(pp(2), Q6);
             diff(pp(3), Q1) diff(pp(3), Q2) diff(pp(3), Q3) diff(pp(3), Q4)
diff(pp(3), Q5) diff(pp(3), Q6)];
        J = double(subs(J,[Q1 Q2 Q3 Q4 Q5 Q6], mp.nodes(vid, :)));

        Psub = subs(pp, [Q1 Q2 Q3 Q4 Q5 Q6], mp.nodes(vid, :));
        d = G - eval(Psub);
        d = d/norm(d);
        u = J'*d;
        u = u*dstep;
        params.robot = mp.nodes(vid, :) + u'; % New configuration

        n                      = size(mp.nodes,1);
```

```
        mp.nchildren(vid)      = mp.nchildren(vid) + 1;
        mp.nodes(n + 1, :)     = params.robot;
        mp.parents(n + 1)      = vid;
        mp.nchildren(n + 1)    = 0;

        if HasRobotReachedGoal() == 1
            mp.vidAtGoal = n + 1;
            return;
        end

        iter = iter + 1;
        vid = vid + 1;
        if mod(iter, 20) == 0
            fprintf('Iteration = %g\n', iter);
        end
    end

    if mp.vidAtGoal >= 1
        JointTrajectory  = MPGetPath();
        JointTrajectory_smooth = SmoothPath(JointTrajectory);
    end
    %output the output moves
    OutputMovesForUR(JointTrajectory_smooth, 'PotentialField.script');
    Draw(JointTrajectory_smooth);
end
```

DHTransformation.m

```
function T = DHTransformation(Qrobot, t)
    syms Q1 Q2 Q3 Q4 Q5 Q6
    alphaa=[90, 0, 0, 90, -90, 0]; % this is the alpha value for all the
links
    a=[0, -0.425, -0.39225, 0, 0, 0]; % Length of the Link
    d=[0.8916, 0, 0, 0.10915, 0.09456, 0.0823]; %Offset
    Q=[Q1 Q2 Q3 Q4 Q5 Q6]; % joint angle variation
    %%Transformation Matrices
    for i=1:6
        switch i
            case 1
                T01= [cos(Q(1,i)),-
sin(Q(1,i))*cosd(alphaa(1,i)),sind(alphaa(1,i))*sin(Q(1,i)),a(1,i)*cos(Q(1,i)
);sin(Q(1,i)),cos(Q(1,i)).*cosd(alphaa(1,i)),-
sind(alphaa(1,i))*cos(Q(1,i)),sin(Q(1,i))*a(1,i);0,sind(alphaa(1,i)),cosd(alp
haa(1,i)),d(1,i);0,0,0,1];
            case 2
                T12= [cos(Q(1,i)),-
sin(Q(1,i))*cosd(alphaa(1,i)),sind(alphaa(1,i))*sin(Q(1,i)),a(1,i)*cos(Q(1,i)
);sin(Q(1,i)),cos(Q(1,i)).*cosd(alphaa(1,i)),-
sind(alphaa(1,i))*cos(Q(1,i)),sin(Q(1,i))*a(1,i);0,sind(alphaa(1,i)),cosd(alp
haa(1,i)),d(1,i);0,0,0,1];
            case 3
```

```
                T23= [cos(Q(1,i)),-
sin(Q(1,i))*cosd(alphaa(1,i)),sind(alphaa(1,i))*sin(Q(1,i)),a(1,i)*cos(Q(1,i)
);sin(Q(1,i)),cos(Q(1,i)).*cosd(alphaa(1,i)),-
sind(alphaa(1,i))*cos(Q(1,i)),sin(Q(1,i))*a(1,i);0,sind(alphaa(1,i)),cosd(alp
haa(1,i)),d(1,i);0,0,0,1];
            case 4
                T34= [cos(Q(1,i)),-
sin(Q(1,i))*cosd(alphaa(1,i)),sind(alphaa(1,i))*sin(Q(1,i)),a(1,i)*cos(Q(1,i)
);sin(Q(1,i)),cos(Q(1,i)).*cosd(alphaa(1,i)),-
sind(alphaa(1,i))*cos(Q(1,i)),sin(Q(1,i))*a(1,i);0,sind(alphaa(1,i)),cosd(alp
haa(1,i)),d(1,i);0,0,0,1];
            case 5
                T45= [cos(Q(1,i)),-
sin(Q(1,i))*cosd(alphaa(1,i)),sind(alphaa(1,i))*sin(Q(1,i)),a(1,i)*cos(Q(1,i)
);sin(Q(1,i)),cos(Q(1,i)).*cosd(alphaa(1,i)),-
sind(alphaa(1,i))*cos(Q(1,i)),sin(Q(1,i))*a(1,i);0,sind(alphaa(1,i)),cosd(alp
haa(1,i)),d(1,i);0,0,0,1];
            case 6
                T56= [cos(Q(1,i)),-
sin(Q(1,i))*cosd(alphaa(1,i)),sind(alphaa(1,i))*sin(Q(1,i)),a(1,i)*cos(Q(1,i)
);sin(Q(1,i)),cos(Q(1,i)).*cosd(alphaa(1,i)),-
sind(alphaa(1,i))*cos(Q(1,i)),sin(Q(1,i))*a(1,i);0,sind(alphaa(1,i)),cosd(alp
haa(1,i)),d(1,i);0,0,0,1];
        end
    end

    switch t
        case 1
            T = T01;
        case 2
            T = T01*T12;
        case 3
            T = T01*T12*T23;
        case 4
            T = T01*T12*T23*T34;
        case 5
            T = T01*T12*T23*T34*T45;
        case 6
            T = T01*T12*T23*T34*T45*T56;
        otherwise
            T = [];
    end
end
```

Sending Path to UR5

OutputMovesForUR.m

```
%this method outputs a file with all the moves for UR software

function OutputMovesForUR(path, filename)
    %open the file
    file = fopen(filename, 'w');
    %output frames from begin to end (0->1)
```

```
    for key = 1 : size(path,1)
        fprintf(file, 'movej([%g, %g, %g, %g, %g, %g], a=1, v=1, t=0,
r=0)\n', path(key, 1), path(key, 2), path(key, 3), path(key, 4), path(key,
5), path(key, 6));
    end
    %wait for 2 seconds
    fprintf(file, 'sleep(2.0)\n');
    %output frames from end to begin (1->0)
    for key = size(path,1) : -1 : 1
        fprintf(file, 'movej([%g, %g, %g, %g, %g, %g], a=1, v=1, t=0,
r=0)\n', path(key, 1), path(key, 2), path(key, 3), path(key, 4), path(key,
5), path(key, 6));
    end
    %wait for 2 seconds
    fprintf(file, 'sleep(2.0)\n');
    %close the file
    fclose(file);
end
```

References

Akkaya, N. Path Finding using Rapidly-Exploring Random Tree. Retrieved 15 August 2019, from https://nakkaya.com/2011/10/27/path-finding-using-rapidly-exploring-random-tree/

Alandes, C. (2015). *A comparison among different sampling-based planning techniques* (Master of Science in Automation and Control Engineering). POLITECNICO DI MILANO.

Choset, H., Lynch, K., & Hutchinson, S. (2004). *Principles of robot motion* (pp. 233-235). Massachusetts: The MIT Press.

Choset, H., Lynch, K., & Hutchinson, S. (2004). *Principles of robot motion* (pp. 213-216). Massachusetts: The MIT Press.

Robot. (n.d.) In *Merriam-Webster's collegiate dictionary*. Retrieved from https://www.merriam-webster.com/dictionary/robots

Path Planning—Artificial Potential Field Method (2017). Retrieved from https://blog.csdn.net/xiaoma_bk/article/details/78507750