COMP0127 Robotics System Engineering Coursework 2

Hou Nam Chiang 15055142

22nd December, 2020

Question 1

There are 6 inverse kinematic solutions exist for a 2D 4 revolute joint manipulator if an achievable pose of the end-effector x_e is given. The pose of the endpoint and the first joint is fixed. So there are only 2 joints that can move to different positions. So two of them together have 4 configurations and therefore 4 inverse KE solutions.

Question 2

Since for this question, we have more than one inverse solution exist for the desired pose of the end effector x_e, we have to consider the solvability in terms of robot workspace and joint limits. For instance, if the solution requires one joint or more hanging out of the workspace or a boundary such as a wall, we should not use this solution. In terms of joint limits, every joint of a robot should have their own joint limits defined by the manufacturer, if the solution requires any of these joints to rotate at an angle outside of these limits, we should not use this solution

Question 3

For the output of the function atan2(x, y) and atan($\frac{x}{y}$), these two functions will give a different output when $y < 0$, this help us to identify which quadrant the coordinate (y, x) is in.

Question 4

(a) To compute the Jacobian matrix $\mathbf{J}$ for the Youbot manipulator, since all the joints are revolute joints, we can calculate $\mathbf{J}$ as follows:

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_{P_1} & \dots & \mathbf{J}_{P_i} & \dots & \mathbf{J}_{P_n} \\ \mathbf{J}_{O_1} & \dots & \mathbf{J}_{O_i} & \dots & \mathbf{J}_{O_n} \end{bmatrix}$$

with:

$$\mathbf{J}_{P_i} = \mathbf{z}_{i-1}^0 \times (\mathbf{p}_e^0 - \mathbf{o}_{i-1}^0)$$
$$\mathbf{J}_{O_i} = \mathbf{z}_{i-1}^0$$

where:

(a) $\mathbf{p}_e^0$ is the first to third entries of 4^{th} column of ${}^0T_e$.

(b) $\mathbf{o}_{i-1}^0$ is the first to third entries of 4^{th} column of ${}^0T_{i-1}$.

(c) $\mathbf{z}_{i-1}^0$ is is the first to third entries of 3^{rd} column of ${}^0T_{i-1}$.

The code that compute the above are as follows:

```python
def get_jacobian(self, joint):
    ##TODO: Fill in this function to complete the question 4a
    J     = np.zeros((6,5)) # Initialize Jacobian
    T_All = []              # Memory for transformation matrices

    # Forward KE to calculate Transformation matrices
    for frame in range(0,5):
        T_All.append(self.forward_kine(joint, frame))

    # p_e^0:
    p_e = T_All[len(T_All)-1][0:3,3]

    for i in range(0,5):
        # Select i th transformation matrix:
        Tr      = T_All[i]
        z_previ = Tr[0:3,2]
        o_previ = Tr[0:3,3]

        # J_Pi:
        J[0:3,i] = np.cross(z_previ, (p_e - o_previ))

        # J_Oi:
        J[3:6,i] = z_previ

    return J
```

(b) To derive the closed-form inverse kinematics solutions for the YouBot, we first need to find ${}^0T_5$ using the DH parameters:

$$\begin{aligned}
{}^0T_5 &= \prod_{i=1}^{n} {}^{i-1}T_i \\
&= \prod_{i=1}^{n} \begin{bmatrix} cos(\theta_i) & -sin(\theta_i)cos(\alpha_i) & sin(\theta_i)sin(\alpha_i) & a_i cos(\theta_i) \\ sin(\theta_i) & cos(\theta_i)cos(\alpha_i) & -cos(\theta_i)sin(\alpha_i) & a_i sin(\theta_i) \\ 0 & sin(\alpha_i) & cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_{ex} \\ r_{21} & r_{22} & r_{23} & p_{ey} \\ r_{31} & r_{32} & r_{33} & p_{ez} \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}$$

where:

$$\begin{aligned}
r_{11} =& sin(\theta_1)sin(\theta_5) + cos(\theta_2+\theta_3+\theta_4)cos(\theta_1)cos(\theta_5) \\
r_{12} =& cos(\theta_2+\theta_3+\theta_4)cos(\theta_1)sin(\theta_5) - cos(\theta_5)sin(\theta_1) \\
r_{13} =& sin(\theta_2+\theta_3+\theta_4)cos(\theta_1) \\
r_{21} =& cos(\theta_2+\theta_3+\theta_4)cos(\theta_5)sin(\theta_1) - cos(\theta_1)sin(\theta_5) \\
r_{22} =& cos(\theta_1)cos(\theta_5) + cos(\theta_2+\theta_3+\theta_4)sin(\theta_1)sin(\theta_5) \\
r_{23} =& sin(\theta_2+\theta_3+\theta_4)sin(\theta_1) \\
r_{31} =& -sin(\theta_2+\theta_3+\theta_4)cos(\theta_5) \\
r_{32} =& -sin(\theta_2+\theta_3+\theta_4)sin(\theta_5) \\
r_{33} =& cos(\theta_2+\theta_3+\theta_4) \\
p_{ex} =& \frac{37}{200}sin(\theta_2+\theta_3+\theta_4)cos(\theta_1) + \frac{1}{1000}cos(\theta_1)(2cos(\theta_2+\theta_3+\theta_4) + 135sin(\theta_2+\theta_3)... \\
&+155sin(\theta_2)+33) \\
p_{ey} =& \frac{37}{200}sin(\theta_2+\theta_3+\theta_4)sin(\theta_1) + \frac{1}{1000}sin(\theta_1)(2cos(\theta_2+\theta_3+\theta_4) + 135sin(\theta_2+\theta_3)... \\
&+155sin(\theta_2)+33) \\
p_{ez} =& \frac{37}{200}cos(\theta_2+\theta_3+\theta_4) - \frac{1}{500}sin(\theta_2+\theta_3+\theta_4) + \frac{27}{200}cos(\theta_2+\theta_3) + \frac{31}{200}cos(\theta_2) + \frac{29}{200}
\end{aligned}$$

These equations will be used to calculate the inverse kinematics. Using p_{ex}, p_{ey}, p_{ez}:

$$\begin{aligned}
\frac{p_{ex} - \frac{37}{200}r_{13}}{cos(\theta_1)} =& \frac{1}{1000}(2r_{33} + 135sin(\theta_2+\theta_3) + 155sin(\theta_2) + 33) \\
\frac{p_{ey} - \frac{37}{200}r_{23}}{sin(\theta_1)} =& \frac{1}{1000}(2r_{33} + 135sin(\theta_2+\theta_3) + 155sin(\theta_2) + 33) \\
p_{ez} - \frac{37}{200}r_{33} + \frac{1}{500}\sqrt{1-r_{33}^2} - \frac{29}{200} =& \frac{27}{200}cos(\theta_2+\theta_3) + \frac{31}{200}cos(\theta_2)
\end{aligned}$$

We first get:

$$\theta_1 = atan2\left(p_{ey} - \frac{37}{200}r_{23}, p_{ex} - \frac{37}{200}r_{13}\right)$$

Let $k = \frac{p_{ex} - 0.185 r_{13}}{cos\theta_1}$ if $sin\theta_1 = 0$, else $k = \frac{p_{ey} - 0.185 r_{23}}{sin\theta_1}$. we have:

$$\begin{aligned}
k - \frac{1}{500} r_{33} - \frac{33}{1000} &= \frac{27}{200} sin(\theta_2 + \theta_3) + \frac{31}{200} sin(\theta_2) \qquad (1)\\
k - \frac{1}{500} r_{33} - \frac{33}{1000} &= \frac{27}{200} sin(\theta_2 + \theta_3) + \frac{31}{200} sin(\theta_2) \qquad (2)\\
p_{ez} - \frac{37}{200} r_{33} + \frac{1}{500}\sqrt{1 - r_{33}^2} - \frac{29}{200} &= \frac{27}{200} cos(\theta_2 + \theta_3) + \frac{31}{200} cos(\theta_2) \qquad (3)
\end{aligned}$$

Then, using (1) and (3), we have:

$$\begin{aligned}
&(k - 0.002 r_{33} - 0.033)^2 + \left(p_{ez} - 0.185 r_{33} + 0.002\sqrt{1 - r_{33}^2} - 0.145\right)^2 ...\\
&\qquad = 0.04225 + 0.04185(sin(\theta_2 + \theta_3) sin\theta_2 + cos(\theta_2 + \theta_3) cos\theta_2)\\
&(k - 0.002 r_{33} - 0.033)^2 + \left(p_{ez} - 0.185 r_{33} + 0.002\sqrt{1 - r_{33}^2} - 0.145\right)^2 = 0.04225 + 0.04185 cos(\theta_3)\\
&K = 0.04225 + 0.04185 cos(\theta_3)\\
&\theta_3 = cos^{-1}\left(\frac{K - 0.04225}{0.04185}\right)
\end{aligned}$$

Then back in (1), we can have:

$$\begin{aligned}
k - \frac{1}{500} r_{33} - \frac{33}{1000} &= \frac{27}{200} sin(\theta_2 + \theta_3) + \frac{31}{200} sin(\theta_2)\\
&= 0.135(sin\theta_2 cos\theta_3 + cos\theta_2 sin\theta_3) + 0.155 sin\theta_2\\
&= (0.135 cos\theta_3 + 0.155) sin\theta_2 + 0.135 cos\theta_2 sin\theta_3\\
&= A sin\theta_2 + B cos\theta_2\\
\frac{k - \frac{1}{500} r_{33} - \frac{33}{1000}}{\sqrt{A^2 + B^2}} &= sin(\theta_2 + \gamma)
\end{aligned}$$

and back in (3), we have:

$$\begin{aligned}
p_{ez} - \frac{37}{200} r_{33} + \frac{1}{500}\sqrt{1 - r_{33}^2} - \frac{29}{200} &= 0.135 cos(\theta_2 + \theta_3) + 0.155 cos(\theta_2)\\
&= 0.135(cos\theta_2 cos\theta_3 - sin\theta_2 sin\theta_3) + 0.155 cos\theta_2\\
&= (0.135 cos\theta_3 + 0.155) cos\theta_2 - 0.135 sin\theta_2 sin\theta_3\\
&= A cos\theta_2 - B sin\theta_2\\
\frac{p_{ez} - \frac{37}{200} r_{33} + \frac{1}{500}\sqrt{1 - r_{33}^2} - \frac{29}{200}}{\sqrt{A^2 + B^2}} &= cos(\theta_2 + \gamma)
\end{aligned}$$

where $\gamma = atan2(A, B)$. Then combining the above, we have:

$$\begin{aligned}
\frac{k - \frac{1}{500} r_{33} - \frac{33}{1000}}{p_{ez} - \frac{37}{200} r_{33} + \frac{1}{500}\sqrt{1 - r_{33}^2} - \frac{29}{200}} &= tan(\theta_2 + \gamma)\\
\theta_2 &= atan2\left(k - \frac{1}{500} r_{33} - \frac{33}{1000}, p_{ez} - \frac{37}{200} r_{33} + \frac{1}{500}\sqrt{1 - r_{33}^2} - \frac{29}{200}\right)\\
&\quad ... - atan2(A, B)
\end{aligned}$$

After that, we have:

$$\theta_4 = cos^{-1}(r_{33}) - \theta_2 - \theta_3$$

Then finally, we have:

$$tan\theta_5 = \frac{r_{32}}{r_{31}}$$
$$\theta_5 = atan2(r_{32}, r_{31})$$

(c) To compute the inverse kinematics iteratively, we do the followings:

(a) Initialize looping parameter k = 0

(b) Then repeatly perform the followings until the solution converges:

i. Calculate the forward kinematics for joint positions $\mathbf{q}(k)$ to get the transformation matrices $\mathbf{T}(\mathbf{q}(k))$

ii. Compute the Jacobian matrix $\mathbf{J}(k)$

iii. Compute the pose $\mathbf{x}_e(\mathbf{T}(\mathbf{q}(k)))$

iv. Calculate the joint positions for the $k+1^{th}$ loop:

$$\mathbf{q}(k+1) = \mathbf{q}(k) + \alpha \mathbf{J}(k)^T(\mathbf{x}_e^* - \mathbf{x}_e(\mathbf{T}(\mathbf{q}(k))))$$

where $\mathbf{x}_e^*$ is the desired pose

v. Update the looping parameter $k = k + 1$

(c) Return $\mathbf{q}(k+1)$

The convergence criteria I used is to leverage the distance between the desired pose $\mathbf{p}_e^*$ and the current pose $\mathbf{p}_e$ such that the distance is smaller than a smaller value ε:

$$\|\mathbf{x}_e^* - \mathbf{x}_e\| < \varepsilon$$

The code that do the above algorithm is as follows:

```python
def inverse_kine_ite(self, desired_pose, current_joint):
    ##TODO: Fill in this function to complete the question 4c

    # Assume desire_pose is a 6x1 vector (x_e^*)
    # Assume current_joint is a 5x1 vector q(0)

    # Cast input to np arrays
    desired_pose  = np.array(desired_pose)
    current_joint = np.array(current_joint)

    # Initialize:
    k           = 0
    T_e         = np.identity(4)
    x_e         = np.zeros((6,1))
    x_e_star    = np.zeros((6,1))
    alpha       = 0.1 # Converge rate
    error   = 100000

    # Error thershold:
    epsilon = 0.0001

    while error > epsilon:
        # Forward KE for T(q(k))
        T_e = self.forward_kine(current_joint, 5)

        # Jacobian for q(k)
        J = self.get_jacobian(current_joint)

        # Pose at step k
        p_e = T_e[0:3,3]
        R_e = T_e[0:3,0:3]

        # Convert rotation matrix to rodrigues for x_e:
        [rex, rey, rez]=self.convert_quat2rodrigues(self.rotmat2q
           (R_e))

        # Current x_e
        x_e[0:3] = p_e.reshape(3,1)
        x_e[3]   = rex
        x_e[4]   = rey
        x_e[5]   = rez
        x_e_star = desired_pose

        # Pose information at q_k+1 (q_kp1)
        current_joint = current_joint + ...
            np.squeeze(alpha*np.transpose(J).dot((x_e_star - x_e)
               ))
        # Check error
        error = np.linalg.norm(x_e_star - x_e)

        # Debug
        print("Error: {}".format(error))
        # Next step
        k = k + 1
    print("Done in iteration: {}".format(k))
    return current_joint
```

(d) Singularities occur whenever the determinant of the Jacobian **J**, $\mathbf{J} \in \mathbb{R}^{6\times n}$ is rank deficient, meaning that the determinant of J will be null if the configuration is singular:

$$\begin{cases} det(\mathbf{J}) = 0 & n = 6 \\ det(\mathbf{J}^T\mathbf{J}) = 0 & n \neq 6 \end{cases}$$

So, whenever the determinant of the pose is 0, the pose is singular. The code that compute the above is as follows:

```
def check_singularity(self, current_joint):
    ##TODO: Fill in this function to complete the question 4d
    J = self.get_jacobian(current_joint)
    if J.shape[0] == 6 and J.shape[1] == 6:
        det_J = np.linalg.det(J)
    elif J.shape[0] == 6 and J.shape[1] != 6:
        det_J = np.linalg.det(J.T.dot(J))

    if det_J == 0:
        print("Singular!")
```

Question 5

(a) Differential drive system. It is holonomic since the wheels can be controlled individually so the robot can arrive all positions on the ground. The configuration space is the space without the no-go zone and obstacles including facilities like the elevator in the map.

(b) Since all checkpoints and obstacles are marked, we can first place repulsive fields at the centre of the obstacles which are effective for their size. Then we can place an attractive field on the first checkpoint. After the first checkpoint is reached, switch that attractive field off and place another attractive field on the second checkpoint. This process repeats until the goal is reached. If the ordering of the checkpoints is not concerned, the attractive field will be placed to the nearest checkpoint relative to the robot.

(c) Considering initial and final accelerations, to avoid discontinuities between different segments, when constructing a trajectory:

$$p(t) = p_k + (p_{k+1} - p_k)q(\tau)$$

A 5th order polynomial time function $q(\tau)$ is used:

$$q(\tau) = a_0 + a_1\tau + a_2\tau^2 + a_3\tau^3 + a_4\tau^4 + a_5\tau^5$$

with constraints:

(a) $0 \leq q \leq 1$

(b) $0 \leq \frac{t-t_s}{T} \leq 1$

And null boundaries for initial and final velocities and accelerations:

(a) $q'(t_s) = q'(t_f) = 0$

(b) $q''(t_s) = q''(t_f) = 0$

where T is the trajectory duration, t_s, t_f is the trajectory starting and finishing time respectively.

(d) Since the area has perfect odometry, we can plan the path and trajectory such that it is a straight line whenever there is water.

(e) Spanning Tree (ST) Coverage Algorithm [1], which first impose a grid approximation on the area by placing a node in every 4 cells (Figure 1 (a), (b)). Then start from the initial position, construct an ST by depth first search (Figure 1 (c)). Finally generate a closed path circumnavigating the spanning tree edges which completely covered the area (Figure 1 (d)).

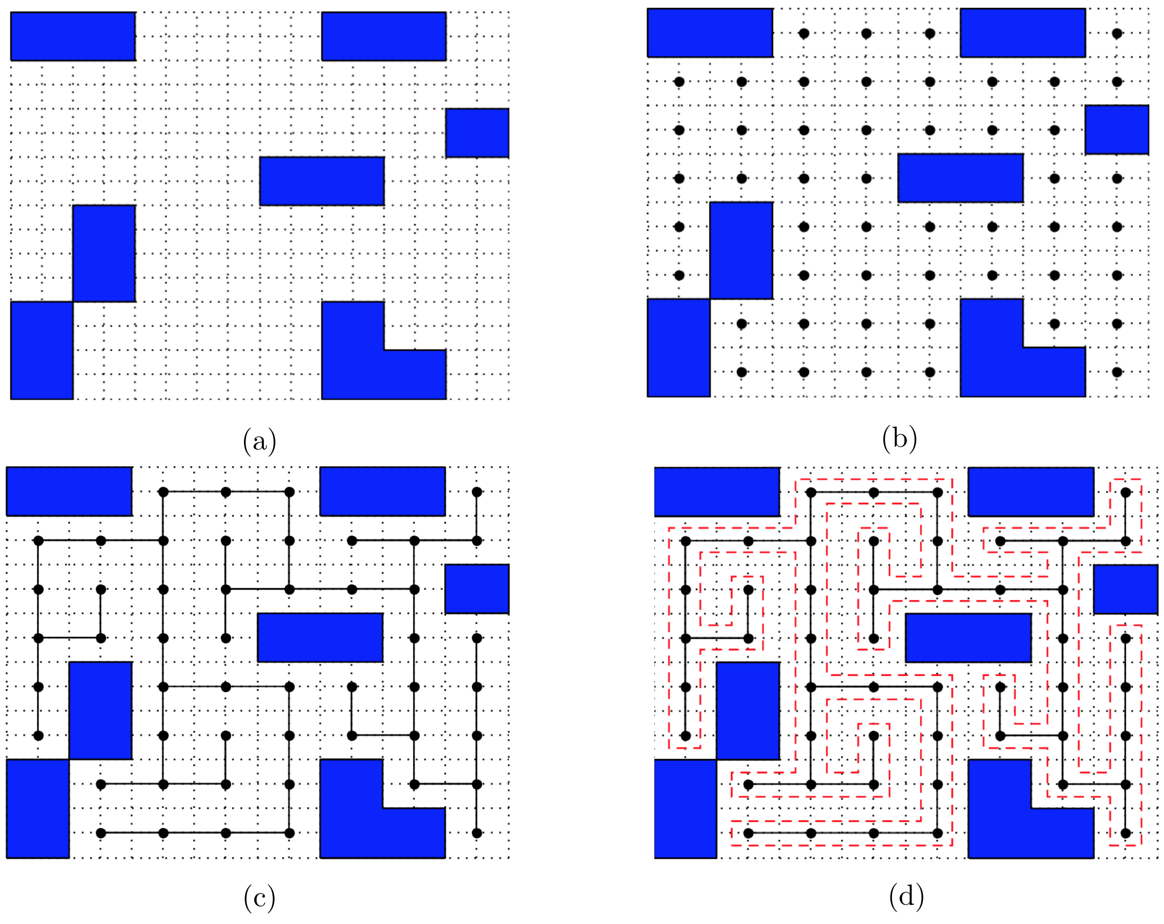


Figure 1: Procedures of Spanning Tree Coverage Algorithm

Question 6

(a) To find the shortest path, a greedy algorithm is used to find the best order of the checkpoints the end effector should visit:

(a) Find the 3D positions of the checkpoints and calculate a cost matrix using the distance between them.

(b) Initialize the scheduled checkpoints with the starting point and the total length to be 0.

(c) Loop until all the checkpoints are visited:

(d) Initialize the minimum distance to the maximum cost (distance) in the cost matrix and the next point = 0.

   i. Loop for all the the unvisited checkpoints:

   ii. Compare the cost (distance) from the last scheduled checkpoint to all other unvisited points with the minimum distance, if it is smaller than the minimum distance, just assign the minimum distance to the cost and the next point to the corresponding checkpoint.

(e) Add the checkpoint to the scheduled checkpoints

(f) Remove the checked points from the unvisited checkpoints.

The code that do the above is as shown, a class is created for the algorithm:

```python
class Greedy(object):
    def __init__(self, cost_matrix, rank):
        self.cost_matrix = cost_matrix
        self.length      = rank

    def greedy_algorithm(self):
        unvisited_checkpoints = list(range(2, self.length + 1))
        scheduled_checkpoints = [1]
        total_distance        = 0

        while len(unvisited_checkpoints) > 0:
            min_distance   = self.cost_matrix.max()
            min_checkpoint = 0

            for checkpoint_idx in range(len(unvisited_checkpoints
               )):
                current_distance =  self.cost_matrix[
                   unvisited_checkpoints[checkpoint_idx] - 1,
                   scheduled_checkpoints[-1] - 1]

                if current_distance == 0 or current_distance <
                   min_distance:
                    min_distance   = current_distance
                    min_checkpoint = checkpoint_idx

            total_distance += min_distance
            scheduled_checkpoints.append(unvisited_checkpoints[
               min_checkpoint])
            unvisited_checkpoints.remove(unvisited_checkpoints[
               min_checkpoint])

        return scheduled_checkpoints
```

After this, a list of ordered indices is returned, and the checkpoints together with the interpolated points are added to the trajectory in that order. Please note that there are checkpoints not turning green even the end effector hit the point. The function used to interpolate between the checkpoints is as shown, KDL is used:

```python
def interpolate(KDL, current_position, point_1, point_2, tfs,
   joint_traj):
    # 1. Interpolate intermediate points:
    n = 5
    x = point_1[0]
    y = point_1[1]
    z = point_1[2]

    for i in range(0, n):
        my_pt = JointTrajectoryPoint()
        # 1.1 Interpolate:
        delta_x = (point_2[0] - point_1[0])/n
        delta_y = (point_2[1] - point_1[1])/n
        delta_z = (point_2[2] - point_1[2])/n
        x = x + delta_x
        y = y + delta_y
        z = z + delta_z
        position = [x,y,z]

        # 2. Inverse KE by KDL for each points to get pose:
        # Desired pose:
        pos = PyKDL.Vector(position[0], position[1], position[2])
        desired_pose = PyKDL.Frame(pos)

        # Save a list of possible pose solutions:
        selected_pose = []

        for i in range(1000):
            pose = KDL.inverse_kinematics_closed(desired_pose)
            selected_pose.append(pose)

        # Compare poses in selected_pose with the
           current_position using Mean squared error:
        mse_all = []
        for pose in selected_pose:
            for i in range(5):
                mse = (pose[i] - current_position[i])**2
            mse /= 5
            mse_all.append(mse)

        current_position = selected_pose[np.argmin(mse_all)]

        # 3. Append to joint_traj
        for j in range(0, 5):
            my_pt.positions.append(selected_pose[np.argmin(
               mse_all)][j])

        my_pt.time_from_start.secs = tfs
        tfs += 60/n
        joint_traj.points.append(my_pt)

    return joint_traj, tfs
```

and the interpolated points are as shown (Figure 2):

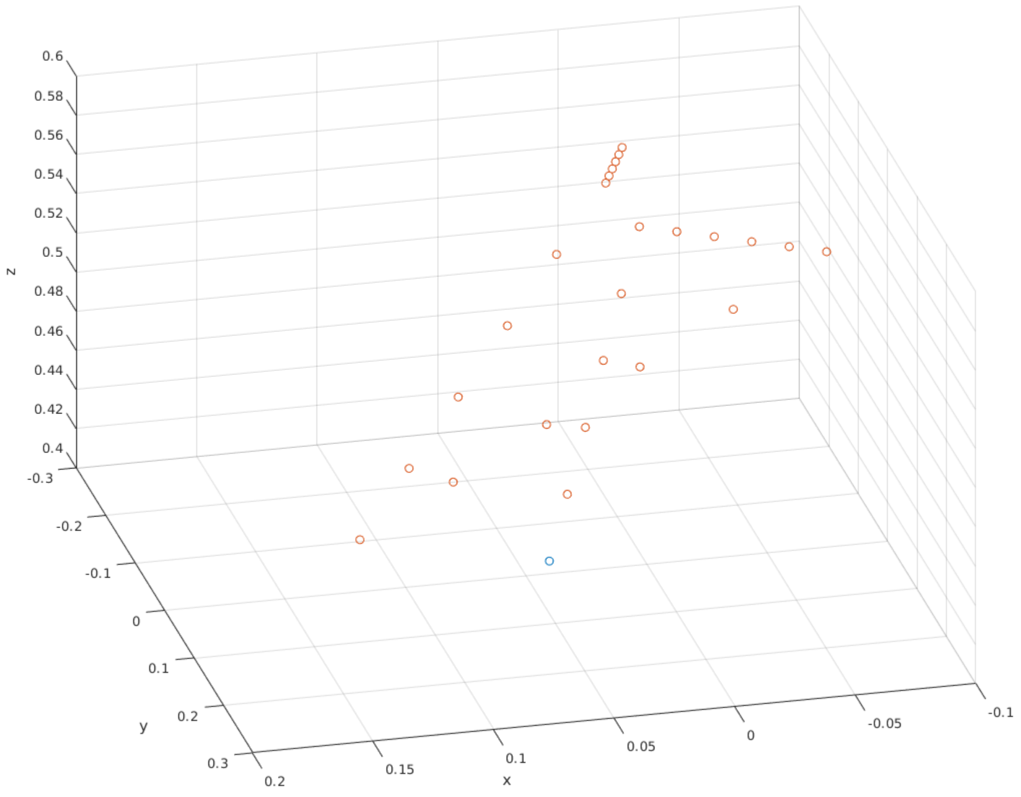


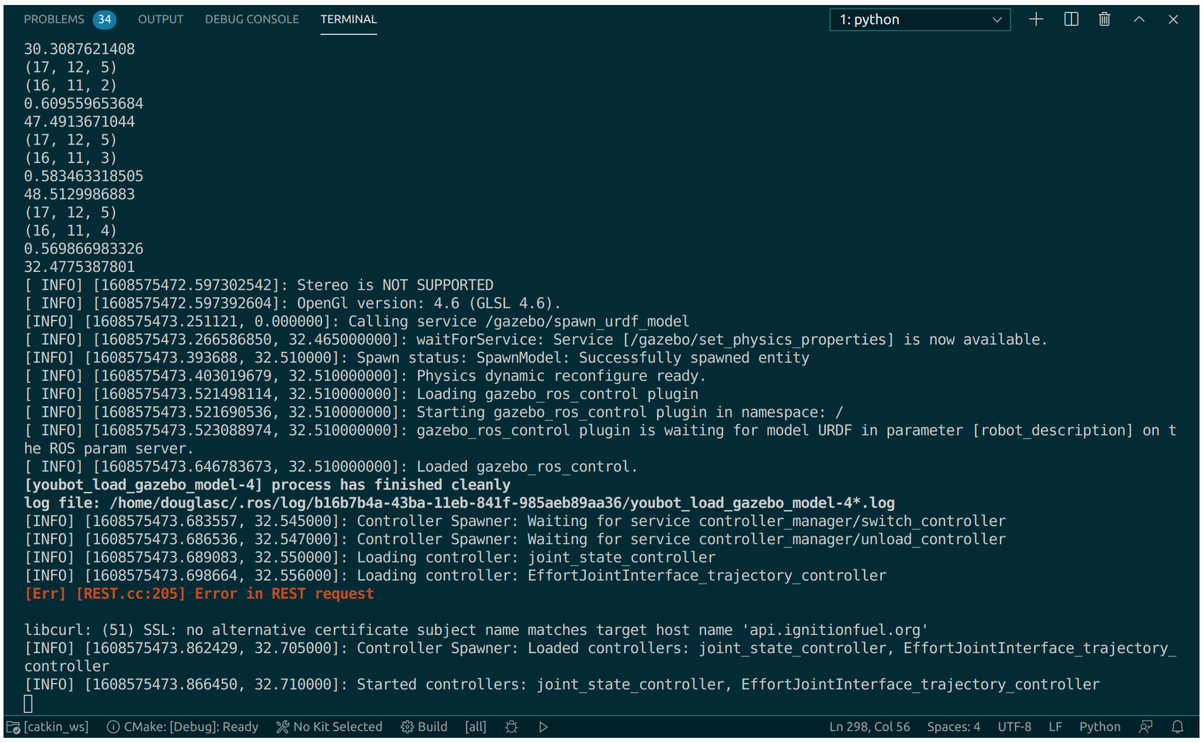
Figure 2

Since the inverse kinematics of KDL do not always return a valid solution, checking is added to the solutions given by the KDL inverse kinematics and those closest to the previous poses are added to the trajectories. This is done using mean square error. With this, the robot arm can move more smoothly between the checkpoints. So please use my uploaded version of KDL (kdl_kine_solver_cw2q6.py).

(b) This task has 5 checkpoints that the end effector has to reach without hitting any obstacles. My idea was to use potential field whenever the end effector travel between two checkpoints, by using the potential field (a class is defined in potential_field.py), the algorithm will return a path that should avoid the obstacles. Efforts have been put in writing the potential field class and the algorithm is as shown:

   (a) Calculate the potential $U = U_a + U_r$ for every point in the region of interest.
   (b) From the starting position, explore the the nearest move around the point by comparing the potentials at these points, and go to the one with the lowest potentials, until it reach the goal position.
   (c) Store the points for the path.

   Then the points in the path are used to get the pose of the robot arm by inverse kinematics and then publish the poses to the trajectory. However, when I was testing with the potential field, it stuck at some point when gazebo / Rviz were initializing (Figure 3):



```
30.3087621408
(17, 12, 5)
(16, 11, 2)
0.609559653684
47.4913671044
(17, 12, 5)
(16, 11, 3)
0.583463318505
48.5129986883
(17, 12, 5)
(16, 11, 4)
0.569866983326
32.4775387801
[ INFO] [1608575472.597302542]: Stereo is NOT SUPPORTED
[ INFO] [1608575472.597392604]: OpenGl version: 4.6 (GLSL 4.6).
[INFO] [1608575473.251121, 0.000000]: Calling service /gazebo/spawn_urdf_model
[ INFO] [1608575473.266586850, 32.465000000]: waitForService: Service [/gazebo/set_physics_properties] is now available.
[INFO] [1608575473.393688, 32.510000]: Spawn status: SpawnModel: Successfully spawned entity
[ INFO] [1608575473.403019679, 32.510000000]: Physics dynamic reconfigure ready.
[ INFO] [1608575473.521498114, 32.510000000]: Loading gazebo_ros_control plugin
[ INFO] [1608575473.521690536, 32.510000000]: Starting gazebo_ros_control plugin in namespace: /
[ INFO] [1608575473.523088974, 32.510000000]: gazebo_ros_control plugin is waiting for model URDF in parameter [robot_description] on t
he ROS param server.
[ INFO] [1608575473.646783673, 32.510000000]: Loaded gazebo_ros_control.
[youbot_load_gazebo_model-4] process has finished cleanly
log file: /home/douglasc/.ros/log/b16b7b4a-43ba-11eb-841f-985aeb89aa36/youbot_load_gazebo_model-4*.log
[INFO] [1608575473.683557, 32.545000]: Controller Spawner: Waiting for service controller_manager/switch_controller
[INFO] [1608575473.686536, 32.547000]: Controller Spawner: Waiting for service controller_manager/unload_controller
[INFO] [1608575473.689083, 32.550000]: Loading controller: joint_state_controller
[INFO] [1608575473.698664, 32.556000]: Loading controller: EffortJointInterface_trajectory_controller
[Err] [REST.cc:205] Error in REST request

libcurl: (51) SSL: no alternative certificate subject name matches target host name 'api.ignitionfuel.org'
[INFO] [1608575473.862429, 32.705000]: Controller Spawner: Loaded controllers: joint_state_controller, EffortJointInterface_trajectory_
controller
[INFO] [1608575473.866450, 32.710000]: Started controllers: joint_state_controller, EffortJointInterface_trajectory_controller
```

Figure 3

   In the screenshot, (17, 12, 5) is the size of the potential map, and (16, 11, 4) is the current loop calculating the potentials, 0.569866983326 is the attractive potential and 32.4775387801 is the repulsive potential. And then the info. from gazebo and Rviz pops up and the terminal did not give any error related to the program.

   So I have devised a strategy as follows:

   (a) Face the robot arm to the checkpoint by just turning the first joint.
   (b) Extend the arm to the checkpoint.
   (c) Retract the arm and then turn to the other checkpoint.
   (d) Repeat until all checkpoints are reached.

The code that do the above is as shown:

```python
def obstable_traj(my_youbot, bag, joint_traj):
    tfs     = 5
    num_msg = 0
    KDL     = robot_kinematic_mine('base_link', 'arm_link_ee')

    joint_traj.header.stamp = rospy.Time.now()
    for i in range(1,6):
        joint_traj.joint_names.append('arm_joint_{}'.format(i))

    checkpoint_frames = []
    initial_position  = my_youbot.current_joint_position
    current_position  = my_youbot.current_joint_position
    # initial_joint_pos = PyKDL.JntArray(5)
    # for i in range(5):
    #     initial_joint_pos[i] = initial_position[i]
    # initial_pose = KDL.forward_kinematics(initial_joint_pos)
    # checkpoint_frames.append(initial_pose)

    # Extract all the msg to checkpoint_frames:
    for topic, msg, t in bag.read_messages(topics=['
       target_position']):
        x = msg.translation.x
        y = msg.translation.y
        z = msg.translation.z
        rot = q2rot(msg.rotation)
        #print([x,y,z])
        pos_kdl  = PyKDL.Vector(x, y, z)
        rot_kdl  = PyKDL.Rotation(rot[0, 0], rot[0, 1], rot[0, 2],
            rot[1, 0], rot[1, 1], rot[1, 2], rot[2, 0], rot[2,
           1], rot[2, 2])
        checkpoint_frame = PyKDL.Frame(rot_kdl, pos_kdl)
        checkpoint_frames.append(checkpoint_frame)
    bag.close()

    for i in range(len(checkpoint_frames)):
        selected_pose = []

        for k in range(1000):
            pose = KDL.inverse_kinematics_closed(
               checkpoint_frames[i])
            selected_pose.append(pose)

        # Compare poses in selected_pose with the
           current_position using Mean squared error:
        mse_all = []
        for pose in selected_pose:
            for j in range(5):
                mse = (pose[j] - current_position[j])**2
            mse /= 5
            mse_all.append(mse)

        current_position = selected_pose[np.argmin(mse_all)]

        my_pt = JointTrajectoryPoint()
        my_pt.positions.append(selected_pose[np.argmin(mse_all)][0])
        for j in range(1, 5):
            my_pt.positions.append(initial_position[j])

        my_pt.time_from_start.secs = tfs
        tfs += 30
        joint_traj.points.append(my_pt)
        num_msg += 1

        my_pt = JointTrajectoryPoint()
        for j in range(0, 5):
            my_pt.positions.append(selected_pose[np.argmin(mse_all)][j])

        my_pt.time_from_start.secs = tfs
        tfs += 30
        joint_traj.points.append(my_pt)
        num_msg += 1

        my_pt = JointTrajectoryPoint()
        my_pt.positions.append(selected_pose[np.argmin(mse_all)][0])
        for j in range(1, 5):
            my_pt.positions.append(initial_position[j])

        my_pt.time_from_start.secs = tfs
        tfs += 30
        joint_traj.points.append(my_pt)
        num_msg += 1
```

Although the checkpoint can be reached, the robot do hit the obstacles.

References

[1] Chunqing Gao, Yingxin Kou, Zhanwu Li, An Xu, You Li and Yizhe Chang. *Optimal Multirobot Coverage Path Planning: Ideal-Shaped Spanning Tree.* Mathematical Problems in Engineering, vol. 2018, Article ID 3436429, 10 pages, 2018.
`https://doi.org/10.1155/2018/3436429`