Logo

Description automatically generated with medium confidence

**Module Title:** Human-Robot Interaction

**Module Code:** EE4705

**Project No.:** 3

**Title of Lab Exercise:** Project 3 Robot Manipulator Collision Avoidance Path Planning

**Module Group No.:** EE4705-6

**Student Contributed to This Assignment:**

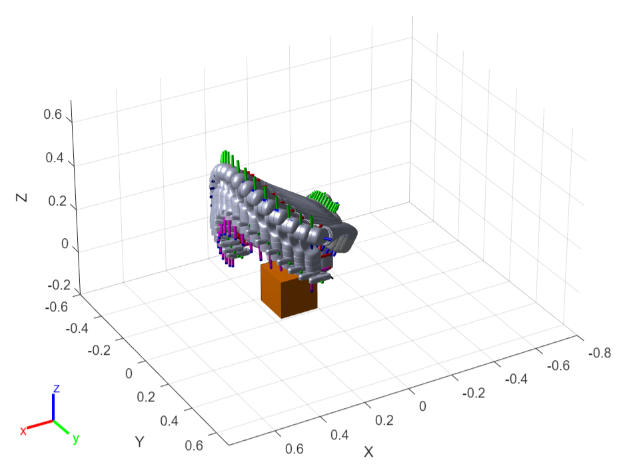
| Admission Number | Name |
| --- | --- |
| A0184893L | Ng Wei Jie, Brandon |
| A220837H | Zheng Jiezhi |
| A0248558M | Zhao Hengyuan |

**Date of Lab Exercise:** 9th April 2022

**Module Lecturer:** Professor Zhao Lin

**T1. Complete the sample code given to you such that upon completion**

The figure belows shows a successful plan collision-free path. Refer to the code under the directory task1 for the implementation.



The transformation matrix for the current joint configurations is implemented below. The getTransform function is used to compute the transformation matrix. The position of the joints are obtained from the 4th columns of the transformation matrix.

for k=1:nJoints

bodyLinkName = bodyLinkNames{k};

T\_matrix = getTransform(robot\_struct, tConfiguration, bodyLinkName, baseLinkName);

T{k} = T\_matrix;

homogenousCoordinates = T\_matrix(:,4);

X(k,:) = homogenousCoordinates;

end

The obstacle cost is computed using the formula given in the STOMP algorithm. The implementation and the formula are also shown below. The safety margin and the sphere radius are added, and the distance from the obstacle from the previous sum. The max of 0 and the previous result is multiplied by the absolute velocity. The sum of the cost array is the obstacle cost.

dxb = voxel\_world\_sEDT(sub2ind(world\_size, idx(:, 1), idx(:, 2), idx(:, 3)));

dist\_array = safety\_margin + radius - dxb;

zero\_array = zeros(size(dist\_array),'like', dist\_array);

cost\_array = max(dist\_array, zero\_array) .\* abs(vel);

cost = sum(cost\_array);

The sampled paths computed using the formula given in the STOMP algorithm. The covariance matrix is used to sample paths around the current mean trajectory. The trajectory at the start and end are kept the same with no deviations.

for m = 1 : nJoints

gau\_dis = mvnrnd(mu, sigma, nSamplePaths);

zero\_dis = zeros(size(gau\_dis(:,1)),'like', gau\_dis);

dis = cat(2,zero\_dis, gau\_dis, zero\_dis);

em{m} = dis;

end

The Stheta is the local trajectory cost obtained from the stompTrajCost function.

Stheta = zeros(nPaths, nDiscretize);

for i=1:nPaths

theta\_path = theta\_paths{i};

[local\_trajectory\_cost, ~] = stompTrajCost(robot\_struct, theta\_path, R, voxel\_world);

Stheta(i,:) = local\_trajectory\_cost;

end

The deviations are computed by the element-wise multiplication of the trajectory probability matrix and the sampled trajectory matrix according to the formula given in the STOMP algorithm. The formula and implementation are shown below.

for i=1:nJoints

em\_i = em{i};

T = trajProb .\* em\_i;

T\_sum = sum(T, 1);

dtheta(i,:) = T\_sum;

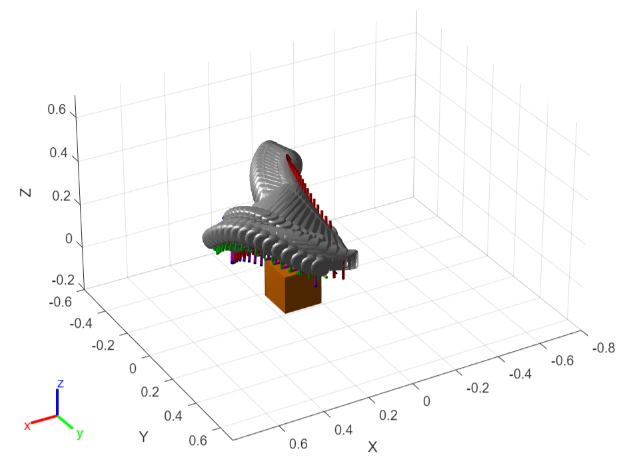
end

The dtheta is smoothened and added to the current mean trajectory according to the formula given in the STOMP algorithm. The formula and implementation are shown below.

[theta, dtheta\_smoothed] = stompUpdateTheta(theta, dtheta, M);

**T2. Choose another robot manipulator for your simulation that is different from the one used in the sample code.**

The robot manipulator chosen is the UR5. The figure shows the robot Refer to the code under the directory task2 for the implementation.

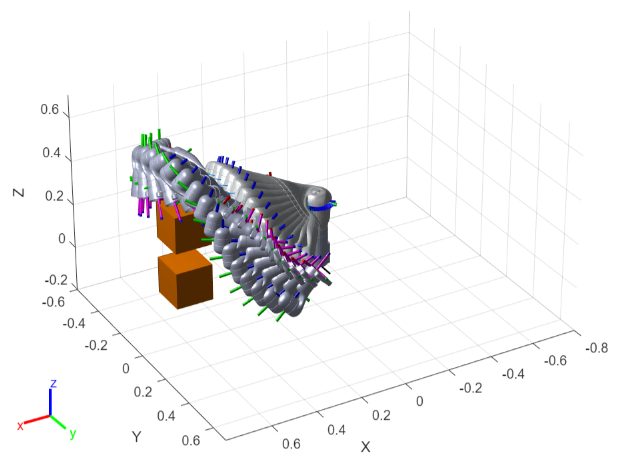


Besides changing the robot name, the index to retrieve the body link name is offset by 2 because the first two joints in this robot are fixed. To prevent the robot from crashing into the obstacle, the safety margin is increased to 0.06. The sphere radius is also increased to 0.06.

**bodyLinkName = bodyLinkNames{k+2};**

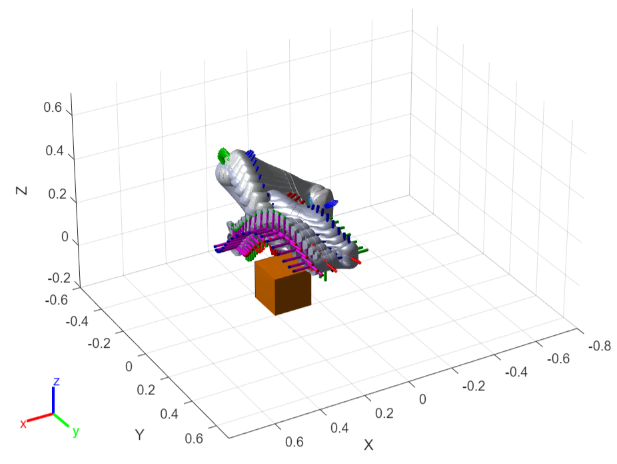
**T3. Create your own collision avoidance scenarios, such as adding additional objects, starting with different initial and final target configurations.**

The robot's initial and final position is changed such that the robot can easily move from source and destination along a straight path. However, the environment is changed such that two obstacles are placed in the middle of the path. After the top obstacle was first added, the robot took a trajectory that went under the top obstacle. After the second obstacle was added, the robot took a trajectory that went above the top obstacle. The STOMP planner computed a feasible path from source to destination after adding new obstacles. Refer to the code under the directory task3 for the implementation.

****

**T4. Add constraint to the robot manipulator end-effector, such that its y-axis is kept upright during moving from the initial position to the target position.**

The figure below shows the robot trajectory after adding the constraint cost to the overall cost such that the y-axis is kept upright. Refer to the code under the directory task4 for the implementation.

****

The formula below is used to compute the constraint cost. The implementation is also shown below.

**T\_d = axang2tform([1 0 0 pi/2])\*axang2tform([0 1 0 pi/2]);**

**T\_e = getTransform(robot\_struct, tConfiguration, endEffectorName, robot\_struct.BaseName);**

**R\_d = T\_d(1:3,1:3);**

**R\_e = T\_e(1:3,1:3);**

**A = eye(3) - transpose(R\_d) \* R\_e;**

**cost = norm(A, "fro") ^ 2;**

The local trajectory cost is computed from the sum of obstacle cost and constraint cost after multiplying by different weights.

**Stheta = 1000\*qo\_cost + 10\*qc\_cost;**

**T5. Replace the MATLAB built-in forward kinematics (getTransform) with programs based-on the twist theory, using the Product of Exponentials (PoE).**

The screw axis is obtained by inspection of each joint frame with respect to the base frame.

**ws = [**

**[0 0 -1];**

**[0 1 0];**

**[0 0 -1];**

**[0 1 0];**

**[0 0 -1];**

**[0 1 0];**

**[0 0 -1];**

**];**

For each joint, the M matrix is computed by multiplying the joint to parent transformation matrix as the “end effector” for the joint with respect to base frame at home position. The velocity is computed by taking the negation of the cross product of the angular velocity and joint position with respect to the base frame. The angular velocity and linear velocity is concatenated and stored in the Slist. The current joint angles are stored in thetalist. The M matrix at each joint is stored in the Ms list.

**M\_ = robot\_struct.Bodies{k}.Joint.JointToParentTransform;**

**M = M \* M\_;**

**homo\_q = M(:,4);**

**q = homo\_q(1:3);**

**w = ws(k, :);**

**v = -cross(w, q);**

**s = cat(2, w, v);**

**Slist(:, k) = s;**

**thetalist(k,:) = tConfiguration(k).JointPosition;**

**Ms{k} = M;**

The original FkinSpace function was modified to compute and store the intermediate matrices without multiplying the M matrix yet. The FkinSpace function repeatedly computes these intermediate matrices for each joint. These are unnecessary computations that slow down the executions.

**function Is = FkinSpaceIntermediates(Slist, thetalist, nJoints)**

**I = eye(4);**

**for i = 1:nJoints**

**I = I \* MatrixExp6(VecTose3(Slist(:, i) \* thetalist(i)));**

**Is{i} = I;**

**end**

**end**

After computing the intermediate matrices, the transformation matrix with respect to the current configuration is obtained by multiplying by the M matrix.

**for k=1:nJoints**

**T\_ = Is{k} \* Ms{k};**

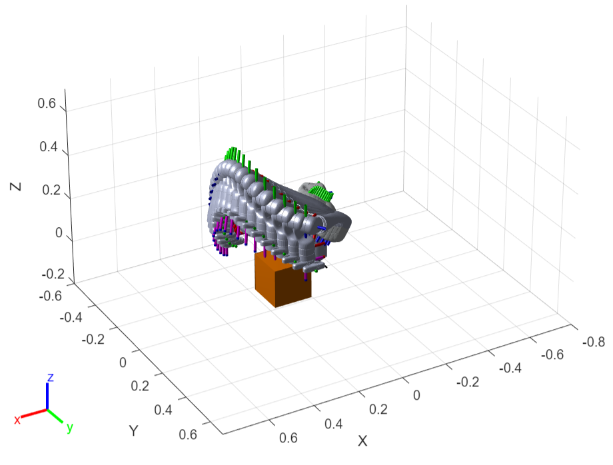
**T{k} = T\_;**

**homo\_q = T\_(:, 4);**

**X(k,:) = homo\_q;**

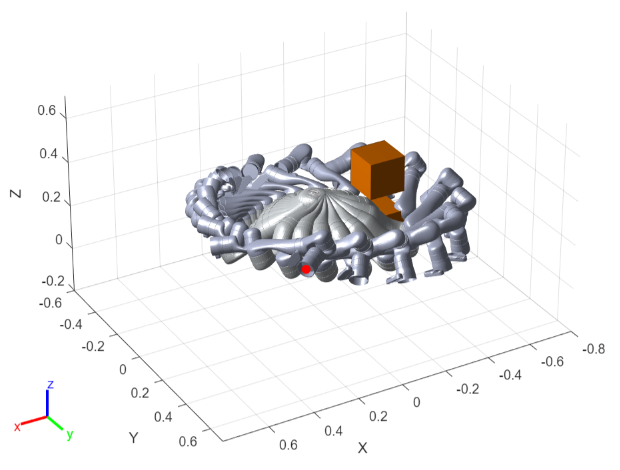
**end**

The figure belows shows a successful plan collision-free path. Refer to the code under the directory task5 for the implementation.

****

**B1. Create a scenario where the STOMP planner fails to plan a feasible trajectory.**

The robot start and end configurations were modified such that the robot swept across the whole plane. Two obstacles are placed to obstruct the trajectory. After the top obstacle was first added, the robot took a trajectory that went under the top obstacle. After the second obstacle was added, the robot failed to take a trajectory that went above the top obstacle. The STOMP planner failed to plan a feasible trajectory. The number of iterations, sphere radius, and marginal error was tuned but the planner failed to plan a feasible trajectory. The STOMP planner uses a trajectory-optimized that uses a signed-distanced approach to minimise the trajectory cost. This allows local optimization. Hence, even though the robot could have taken a trajectory that goes above the top obstacle, the planner is stuck at a local minimum after it went below the obstacle. Refer to the code under the directory bonus1 for the implementation.

****

**B2. Adjust the normalisation/scaling of the covariance matrix, discuss how it affects the planned trajectory. Can you improve the convergence speed of the algorithm by varying the covariance adaptively online?**

Rinv is the covariance matrix and is used as the sigma in the multivariate normal distribution to sample trajectories. A small scaling of the covariance matrix means that the sampled trajectories do not deviate greatly from the current trajectory. A large scaling in the covariance matrix means that the sampled trajectories are varied and the planner does not converge fast.

**Rinv = 1.5\*Rinv/sum(sum(Rinv));**

The large scaling in the covariance matrix is desired to help the planner find a suitable path. When a suitable path is found, a smaller covariance matrix is desired for the planner to converge faster. If the current local trajectory is smaller than the previous, it means the current trajectory is better. The covariance matrix is scaled down. However, a minimal scaling is set to still allow the trajectory to be fine-tuned slightly.

**function [Rinv\_, factor\_] = computeAdaptativeConvarianceMatrix(Rinv, Qtheta, QthetaOld, factor)**

**if Qtheta < QthetaOld**

**if factor < 0.1**

**Rinv\_ = Rinv;**

**factor\_ = factor;**

**else**

**Rinv\_ = 0.9 \* Rinv;**

**factor\_ = 0.9 \* factor;**

**end**

**else**

**Rinv\_ = Rinv;**

**factor\_ = factor;**

**end**

From experimentations, the adaptive covariance matrix allows the planner to converge within 10-16 iterations while the non-adaptive covariance matrix allows the planner to converge within 21-27 iterations on average. The trajectories found using the adaptive covariance matrix seem to be more consistent than those found using the non-adaptive covariance matrix. The covariance matrix sampled trajectories that are more noisy; hence the resultant trajectory is less consistent. Refer to the code under the directory bonus2 for the implementation.

**B3. Create the similar collision avoidance environment as in Task 3 in MoveIt and plan the trajectory using MoveIt version of STOMP. Compare your planning results in both environments and discuss the difference.**

The image below shows that the stomp planner is successfully setup with ROS Moveit. The planning results in both environments in MatLab and MoveIt for Task 3 are very similar. The robot in both environments took a trajectory that went above the top obstacle. The trajectory in MoveIt seems to be slightly higher than MatLab. This is likely due to different safety margins and sphere radius used to compute the obstacle cost. Refer to the code under the directory bonus3 for the implementation.

