# 1 Robot Manipulator Collision Avoidance Path Planning using STOMP

**Notes:**

- This project is of **30% CA**. Report and Codes count for 20% CA. Presentation counts for the other 10% CA. The evaluation will be based on
  - Correctness, quality, efficiency, practical considerations, and additional new features/novel implementation of your code,
  - Challenges and novelty of your design and planning results,
  - Clarity, effectiveness, and visual illustration of your presentations, etc.
  - Your understanding of the theorem/algorithms,

  Please highlight any of the above in your report and representation.
  - Don't make false claims. Be honest.
- This is a group project, but all members should contribute and present individual work. Individual performance in the presentation will be evaluated separately.
- An incomplete sample code will be provided for your group to build upon.

**Tasks:**

1. Complete the sample code given to you such that upon completion, one should be able to run the MATLAB live script `KINOVA_STOMP_Path_Planning.mlx` to successfully plan a collision-free path and show the animation (under the original collision avoidance settings). The following code files are incomplete:`helperSTOMP.m`, `updateJointsWorldPosition.m`, `stompDTheta.m`, `stompSamples.m`, `stompObstacleCost.m`.
2. Choose a different robot manipulator and plan the path for it in a similar scenario to the one in Task 1. On the help page of the MATLAB function `loadrobot()`, you can find a list of robot manipulators that are shipped with Matlab.

   You can also load external robot. Here is a list of the robot manipulators supported in MoveIt: `https://moveit.ros.org/robots/`. The Matlab function `importrobot()` allows you to import an external robot by its URDF. But you also need to specify the mesh file in the command for visualizaiton in Matlab (need some research). Here you can find the xacro and mesh file for Kuka KR210, which has a prismatic joint. You can convert the xacro file to URDF using the ROS command:

   ```
       rosrun xacro xacro model.xacro > model.urdf
   ```

   Use your chosen robotic manipulator for the rest of the tasks.
3. Replace the MATLAB built-in forward kinematics (`getTransform`) with programs based-on the **twist theory**, using the **Product of Exponentials (PoE)** formula. You can use the code provided by the textbook (available on Github, where the function `FKinSpace.m` calculates the forward kinematics given the space twists and end-effector pose in a fixed frame. Note that it depends on other functions in that library). The textbook can be found in this page. But your implementation/integration into your code need to consider code efficiency where necessary. You will need to determine the twists for each axis of your chosen robot according to its geometric parameters, which can be found by checking

the corresponding URDF file, or inspecting the homogeneous transformation matrices associated with each link (see Section 1.1). You should explain clearly how you determine the twists in your report and presentation.

4. Create your collision avoidance scenarios, such as adding additional objects, starting with different initial and final target configurations. Show your path planning results by overlapping the intermediate configurations or using animation. The difficulty and novelty of your scenario will be taken into consideration for evaluation. Do not initialize trivial paths, i.e., collision-free initialization.

5. On top of the previous Tasks, further add constraints to the robot manipulator end-effector, such that it's $y$-axis is kept upright during moving from the initial position to the target position (imaging the task of moving a cup of coffee). You can also choose x or z-axis of the end-effector, and align it with a chosen world frame axis, as long as you can show the difference of the planning results between and after adding the constraints. Since we have fixed the initial and final pose in the code, you may want to adjust the initial and final orientation of the end-effector to align with your desired/chosen orientation to facilitate the planning.

**Remarks**

- You do need to understand all the code to debug and finish all the tasks.
- It is possible that you have to modify the given code to adapt to your own code, for example, adopting a particular data format you defined and used, adjusting the obstacle positions according to the maximum reach of your chosen robot, etc. In addition, if you find the given code inconvenient or inefficient, feel free to modify/improve it. You can also use any external functions, libraries, code snippets, etc.
- Any contribution you make or features you add (e.g., more efficient calculation, adding a GUI for better interaction, fixing a bug in the existing code, improvement/extensions of the STOMP algorithm, etc.) will be considered for grading. Please point it out explicitly in your slides during the demonstration, as well as in your report, if any.

**Submissions**

- Make sure to submit both your PDF report and your MATLAB code. Include all your files into a folder and compress it to a zip file. **Only submit one zip file for your group.**
- **Naming convention**: Group⟨Number⟩_Report_Code.
- **Submission deadline:** Nov. 13th, 2025, 5:00pm.
- For each task, please create a folder containing all necessary codes for reproducing your results. You may want to organize your code according to the 5 tasks, i.e., each folder for each task. You can also submit a single code base, but whatever you submit should make it easy for others to reproduce your solution for all tasks. You don't need to include the code provided by our textbook Modern Robotics (as mentioned in Task 3), if you didn't modify their code. It will be a good practice to include a readme.txt to guide the usage of your code.
- You can use PoE for all tasks in your submission.
- Your presentation serves to explain your work, and your understanding to us. You should explain what you have done, highlight what special features/interesting results you have achieved, etc.

**Presentations**

- **The presentation will be in E1-06-07 on Nov. 13th, Thursday, starting from 6:00PM to 9:00PM. Each group will have 10mins: 7mins for presentation and 3mins for Q&A.**

- You may choose to play a pre-record video for presentation, followed by the Q&A session (all members should present in person).
- **Each member should work on and present a part of your group's work.** You will present in group order, that is, from Group 1 to Group 14 (Please contact TAs if you need rescheduling). Your group only needs to join the meeting a few minutes before your scheduled presentation time and leave after the presentation.
- You can use your own laptop or the in-house teaching desktop for presentations.
- The whole presentation will be recorded (though we will meet in person).

## 1.1 Technical Tips

- The given code relies on the **MATLAB Robotics System Toolbox**. Make sure it is selected when you install the MATLAB.
- Webpage for how to use the `getTransform()` function. **Note** that when read the body names of the robot, Line 1 is much faster than Line 2.

```
1 robot_struct.BodyNames{k}   //fast
2 robot_struct.Bodies{k}.Name //slow
```

- Note that the robot bodies/links may not always start with revolute joints. For UR or some other robots, the bodies/links start with fixed joints that cannot rotate or move. Make sure that you get the positions of the moving joint, which correspond to your joint angle planning correctly.
- For some robots, the URDF file defines the origin of 'base_link' differently from the base frame. Make sure you consistently refer to the same fixed/base/space frame. For example,

```
// with respect to the base frame
getTransform(robot_struct, robot_struct.homeConfiguration, robot_struct.BodyNames{2})

// with respect to the 'base_link', may not coincide with base frame
getTransform(robot_struct, robot_struct.homeConfiguration, robot_struct.BodyNames{2},
'base_link')
```

- To find the robot's geometric parameters, you can load the robots:

```
>> robot_name = 'kinovaGen3';
robot = loadrobot(robot_name, 'DataFormat', 'column');
show(robot);
```

and then check the homogeneous transformation matrix. For example, the following shows the $T_{01}$, which is the configuration of the Joint 1 of KINOVA GEN3 to Joint 1's **parent frame**:

```
>> robot.Bodies{1,1}.Joint.JointToParentTransform
ans =
1.0000      0          0          0
 0    -1.0000     0.0000          0
 0    -0.0000    -1.0000     0.1564
 0          0          0     1.0000
```

- Alternatively, you can refer to the robot's URDF for its geometric information. The following lines shows where the URDF used by MATLAB is retrieved (the source URDF information):

```
>> [gen3Model, robotData]=loadrobot('kinovaGen3')
>> robotData.FilePath    //the local path to the URDF of KinovaGen3
```

Open the URDF, you can find the following snippets of the description of two joints. The left code snippet is for the first joint, named 'Actuator1'. It's geometric data is consistent with the information we found before: the `rpy="3.1416 0 0"` is the Euler angle (roll, pitch, yaw) representation of the orientation of Joint 1 with respect to its parent link (i.e., the base_link), that is, if you rotate the base_link frame around its $x$-axis by $3.1416$ radian, then you will get the orientation of the Joint 1's frame. The `xyz="0 0 0.15643"` shows the translation of the origin of the Joint 1 wrt its parent link (the origin of base_link frame) as well, expressed in base_link's frame. **Note** that the `axis xyz=‘‘0 0 1"` means that the rotational axis of this joint is the z-axis of its own frame (**Not wrt the base frame**).

The right code snippet is for the 2nd joint, named “Actuator2”. Again, the data in origin is with respect to its parent link (i.e., “the Shoulder_Link”). Similiarly, the axis $(0, 0, 1)$ is wrt its **own frame**, **not the base/fixed frame**.

```
<joint
  name="Actuator1"
  type="continuous">
  <origin
    xyz="0 0 0.15643"
    rpy="3.1416 0 0" />
  <parent
    link="base_link" />
  <child
    link="Shoulder_Link" />
  <axis
    xyz="0 0 1" />
  <limit
    effort="39"
    velocity="0.8727" />
</joint>
```

```
<joint
  name="Actuator2"
  type="revolute">
  <origin
    xyz="0 0.005375 -0.12838"
    rpy="1.5708 0 0" />
  <parent
    link="Shoulder_Link" />
  <child
    link="HalfArm1_Link" />
  <axis
    xyz="0 0 1" />
  <limit
    lower="-2.2000"
    upper="2.2000"
    effort="39"
    velocity="0.8727" />
</joint>
```

- **Note that you only need to calculate the screw axes once throughout the whole program.** Because they only depend on the home configuration. When update joints' coordinates, you simply pass the screw axes to the corresponding PoE formulae without recomputing the screw axes all over again.

1. You can also get the rotational axis and positional vector at the home configuration, by using the `getTransform()` function. Then follows the 2D planar car example we have discussed in the class to calculate the screw axis, which simply uses a cross-product. But make sure that your angular velocity and linear velocity are correct.
2. Moreover, you can also use `getTransform()` to get the configuration/pose of a joint **at some arbitrary non-zero angle**. Then use the matrix logarithm map to compute the corresponding twist, and then normalize the twist to get the screw axis, in a way that you have worked out in the last assignment.

- To add constraint for the orientation of the end-effector in Task 5 (i.e., keep the y-axis upright, you can also choose to keep other axis upright), you can either
  1. choose an arbitrary orientation (rotation matrix), whose y-axis is upright wrt the world frame, and add a penalty (i.e., the constraint cost) to the cost function.
  2. Or, you can penalize the difference between the y-axis of the end-effector (the world coordinate of it, denote by $\hat{y}_{sb}$, which you can get from the forward kinematics) and the z-axis of the world frame (which is simply $\hat{z}_s = [0; 0; 1]$) only. For example, the penalty can be the so called $l_1$ norm of the residual vector: $\|\hat{y}_{sb} - \hat{z}_s\|_1 := \sum_{i=1}^{3} |\hat{y}_{sb}(i) - \hat{z}_s(i)|$, which is simply the sum of the absolute values of each element in the residual vector. The $l_1$ norm can be calculated using the `norm(·, 1)` function, where the argument $1$ denotes the $l_1$ norm.

- Multi-variate Gaussian sampling function `mvnrnd` requires the **Matlab Statistics and Machine Learning Toolbox**. Make sure you choose it when you install Matlab.

- Initialization of the joint angles can be influential, especially when there is a discontinuity/singularity. You might need to deal with the discontinuity of the joint angles $\theta \in [-\pi, \pi]$ to improve the initialization, thus improving the trajectory quality. You may also want to check the spherical linear interpolation function `slerp()` provided in MATLAB, which is however based on the quaternion representation of rotations.