Basic Task Planning in Manipulators using AnyGrasp and PointSAM

Kartik Vij, Arya Topale, Pavan Karke

May 2025

Introduction

This project explores the integration of perception, planning, and control for robotic manipulation using a 7-DoF xArm7 robotic arm. The goal is to autonomously identify, plan, and execute a grasp on a specific object (an orange box) in a cluttered environment, simulated using PyBullet. The system incorporates modern 3D perception, grasp pose estimation, motion planning, and frame transformation techniques to bridge the gap between vision and action.

The full codebase is available on GitHub:
GitHub
The core components of the system are:

- **Intel RealSense D455 Camera**: Used to capture RGB-D data and generate a point cloud of the workspace.
- **Point-SAM**: A segmentation model used to extract the target object from the scene.
- **AnyGrasp**: A 7-DoF grasp pose estimation framework that predicts grasp candidates from partial-view point clouds.
- **xArm7 Robot (Simulated in PyBullet)**: A 7-DoF manipulator that executes planned motions.
- **PyBullet-Planning and RRT-Connect**: A motion planning library that computes collision-free trajectories from the current pose to the target grasp pose.
- **ROS and TF**: Used for real-time transformation handling and system integration.

The project pipeline begins with capturing the scene using the RealSense camera, followed by object segmentation via Point-SAM. The segmented point cloud is passed to AnyGrasp to generate feasible grasp poses. These poses,

initially in the camera coordinate frame, are transformed into the robot's frame using hand-eye calibration. Subsequently, a collision-free trajectory is computed using RRT-based planning, and the robot executes the trajectory to successfully grasp the object.

This system demonstrates an end-to-end robotic perception and manipulation pipeline, showcasing the coordination between learned perception models and classical planning algorithms in a realistic simulation environment.

1 Point Cloud Acquisition and Data Refinement

The first stage of the project involves the acquisition of 3D point cloud data, which was carried out using an Intel RealSense D-455 depth camera. This camera captures both color and depth information, allowing for the reconstruction of a 3D representation of a real-world scene. The point cloud serves as the foundational input for all subsequent steps in the grasp planning pipeline.

However, raw point cloud data acquired from depth sensors is often noisy and misaligned, especially in real-world conditions. Factors such as sensor inaccuracies, occlusions, lighting conditions, and surface reflectivity can introduce substantial errors in the spatial representation of objects. These issues can significantly degrade the performance of downstream modules such as segmentation and grasp planning.

To address this, we employed the Iterative Closest Point (ICP) algorithm for point cloud refinement and alignment. ICP is a widely-used algorithm that minimizes the difference between two point clouds by iteratively refining the transformation (rotation and translation) required to best align them. In our context, the ICP algorithm was used for two primary purposes: Noise Reduction and Smoothing: By aligning multiple frames of the same scene (e.g., from a moving or slightly jittery camera), ICP helps in averaging out noise and improving the clarity of surface boundaries.

Precise Alignment: When combining point clouds captured from different viewpoints or stitching partial observations, ICP ensures that the data aligns consistently within a common coordinate frame. This is crucial for accurate segmentation and grasp estimation.

After refinement, the enhanced point cloud is cleaner, more coherent, and better represents the true structure of the scene. This ensures that subsequent components like PointSam and AnyGrasp receive high-quality input, leading to more reliable segmentation and grasp generation.

Frame Transformations

The point cloud obtained from the Intel RealSense D-455 was originally in the camera's optical frame. However, for compatibility with simulation and planning frameworks like PyBullet, MoveIt, and Open3D, it was necessary to align the point cloud to match their coordinate conventions.

The coordinate frame of the D-455 follows the RealSense convention:

- $+Z$ pointing forward (out of the lens)
- $+X$ pointing to the right
- $+Y$ pointing downward

In contrast, PyBullet and MoveIt typically assume:

- $+Z$ pointing upward
- $+X$ pointing forward
- $+Y$ pointing to the left

To align the RealSense frame with the simulation environment, we observed that a two-step rotation was sufficient:

1. A rotation of 180° about the Y-axis: $R_y(180^\circ)$
2. Followed by a rotation of 90° about the Z-axis: $R_z(90^\circ)$

The composite transformation applied to the point cloud is therefore:

$$R = R_z(90^\circ) \cdot R_y(180^\circ)$$

This rotation was implemented directly in the code before publishing the transformed point cloud to AnyGrasp or using it in Open3D or MoveIt. The transformation ensures consistency of grasp pose estimation and motion planning across all components of the system.

The transformation logic can be found in the script `xarm_final.py`.

2 Promptable 3D Segmentation with Point-SAM

Summary. The Processed pointcloud, after converting into the world frame, is sent to Point-SAM Framework. Point-SAM is a 3D extension of the 2D Segment Anything Model (SAM) that performs *promptable* segmentation directly on point clouds. Users can provide positive and negative prompt points to the model via the interactive "demo"; the model then predicts one or more probabilistic masks, scores them, and returns the best mask.

The architecture of PointSAM consists of (1) a point cloud encoder that embeds geometric and color features through patching and a transformer backbone, (2) a prompt encoder for point and mask prompts, and (3) a mask decoder that outputs probability of segmentation per point.

Procedure

1. **Upload or capture a point cloud.** Represent the scene as an $N \times 3$ array of (x, y, z) coordinates (and optionally RGB colors).

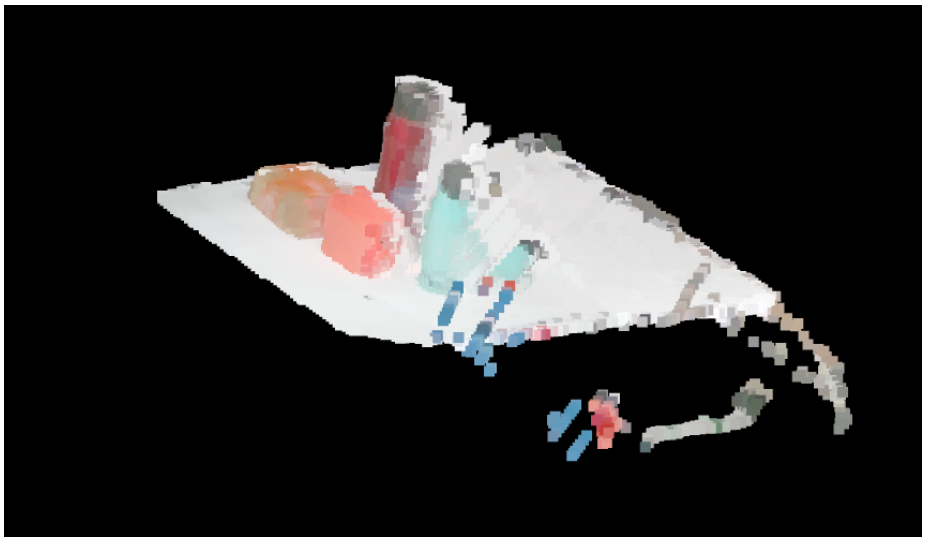

Figure 1: Transformed PointCloud

2. **Normalize and (optionally) down-sample.** Center the cloud at the origin and scale to unit sphere radius; For large N down- sampling of the point cloud is performed to reduce computation cost. The voxel size factor is set accordingly based on the degree to which the point cloud is to be down- sampled. The large voxel size means that more points are retained, leading to larger computation.

3. **Positive prompts.** "Foreground" points, or the points of our target are encoded by clicking on one or more points on the target object. Now the model will attempt to "grow a mask" around them

4. **Negative prompts.** The predicted mask was over-segmenting, i.e., taking more than the required points , hence we supplied appropriate number of "negative" prompt points, which tell the model the points it should not consider.

5. **Mask prediction.** On each prompt iteration, Point-SAM outputs M candidate masks along with predicted IoU scores. The highest-scoring mask is returned and used as the new mask prompt for refinement.

6. **Save or clear.** "Next Mask" is used to commit the current mask and reset prompts for a new object and "Save Annotation" exports all collected masks (e.g. as a NumPy `.npy` file).

2.1 Point-SAM Model Architecture

2.1.1 Point-Cloud Encoder

Given below is what happends behind the scenes:

- **Farthest Point Sampling (FPS).** P *centers* from the cloud via FPS are selected to ensure the point- cloud is well covered.
- **Patch grouping.** For each center, its k nearest neighbors are computed to form a local patch.
- **Patch embedding.** Apply a small PointNet to each patch to extract a feature vector, then add a positional embedding of the center.
- **Transformer backbone.** Feed the P patch embeddings into a Vision Transformer (e.g. Uni3D-Large) to produce a global point-cloud embedding.

2.1.2 Prompt Encoder

- **Point prompts.** Each user-clicked point is represented by its coordinate and a binary label (1=foreground, 0=background). These are projected to the same embedding space and added to learned *label* embeddings.
- **Mask prompts.** The previously predicted mask (a per-point logit) is processed through a small encoder and summed with the point-cloud embedding to refine subsequent predictions.

2.1.3 Mask Decoder

- **Two-way attention.** Just like the Segment Anything Model (SAM), Point-SAM's decoder uses a clever attention mechanism. It has two transformer blocks: one helps the model focus on the user-provided prompt points, and the other helps connect those prompts to the actual point cloud, making the segmentation more accurate.
- **Upsampling.** The model first predicts features for a smaller set of points (called patches), then spreads those predictions back to all original points in the point cloud using a method based on nearby neighbors. This helps the model understand every point in detail.
- **Dynamic classifier.** A small neural network is used to figure out whether each point belongs to the object or not. It uses information from the decoder and applies it to every point to make this decision.
- **Multi-mask outputs.** Sometimes, the model gives several possible segmentations for a single prompt. It also predicts how accurate each option is and selects the one with the best predicted quality.

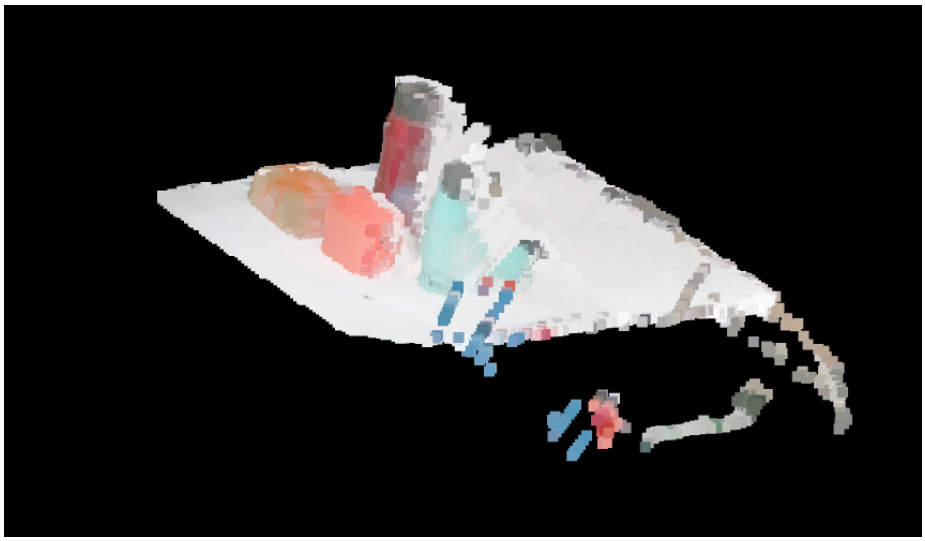

Figure 2: Before Point-SAM Segmentation

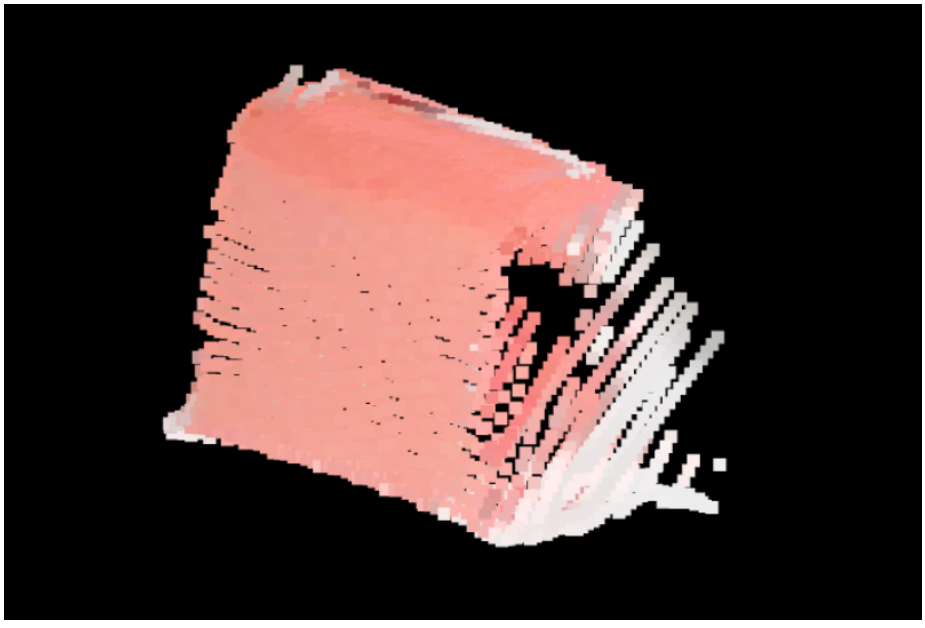

Figure 3: After Point-SAM Segmentation

Figure 4: Point cloud before and after segmentation using Point-SAM.

2.2 Differences from 2D SAM

- **Irregular input.** Point clouds lack the grid structure of images; Point-SAM must group and interpolate dynamically.
- **Scalability.** 3D transformers are more expensive; Point-SAM uses patching and PointNet to reduce complexity.
- **Internal structures.** Native 3D segmentation handles interior geometry (e.g. drawer interiors) that multi-view 2D lifting methods miss.
- **Prompt types.** Both point and mask prompts are directly 3D; no multi-view fusion is required.

3 Grasp Pose Estimation using AnyGrasp

The segmented point cloud of the target object (in our case, the orange box), produced by Point-SAM, is passed into the **AnyGrasp** framework for grasp pose estimation. AnyGrasp is a 7-DoF grasp pose prediction system designed to operate on partial-view point clouds from arbitrary viewpoints. It employs a deep neural network trained to predict high-quality grasp poses across diverse object geometries and occlusion patterns.

In particular, AnyGrasp utilizes a PointNet++-based backbone to extract geometric features from the input point cloud and then regresses multiple grasp pose candidates, each parameterized by a 6-DoF grasp pose and an associated grasp quality score. These candidates are predicted directly from the geometry of the visible surface, without requiring CAD models or complete object reconstructions.

To ensure feasibility, AnyGrasp performs real-time **collision checking** between the predicted gripper poses and the scene point cloud. This step filters out grasp poses that result in collisions with surrounding objects or violate kinematic constraints, ensuring that only valid, executable grasps are passed on.

The final output grasp poses, initially computed in the camera coordinate frame, are transformed into the world frame using the previously established extrinsic calibration matrix. This transformation allows the grasp poses to be interpreted in the robot's base coordinate system, enabling direct use by downstream components such as motion planning and inverse kinematics solvers for grasp execution.

4 Motion Planner

Now, the grasp information (the rotation and the translation) is sent to the motion planner. But before doing that, the grasp, which is in the camera frame, is converted into the world frame. This is done using Hand Eye calibration which will be explained below.

Figure 5: Predicted grasp pose near the segmented orange box, with surrounding objects present in the scene.

4.1 Hand-Eye Calibration

In this project, hand-eye calibration was used to determine the rigid transformation between the end-effector of the xArm7 robotic arm and an externally mounted camera. Unlike traditional approaches that rely on calibration targets like checkerboards or fiducial markers, a simplified method was used due to the fixed and known placement of the camera relative to the robot.

Instead of performing image-based calibration, we utilized ROS tools to iteratively determine the transform. A series of static transforms were manually published using the `static_transform_publisher` node from the `tf2_ros` package to represent the pose of the camera with respect to the robot's end-effector. These transforms were visualized and refined using RViz, where markers or camera feeds were overlaid on the robot model to assess the alignment.

The goal was to achieve a consistent and sufficiently accurate spatial relationship such that 3D positions detected by the camera (e.g., object centroids from depth or RGB-D data) could be transformed correctly into the robot's base frame for downstream tasks such as grasping or manipulation.

This manual calibration approach is feasible and efficient when:

- The camera is rigidly mounted and its pose relative to the robot is approximately known.
- High-precision calibration is not required for the application.
- Real-time feedback in RViz allows iterative refinement by observing alignment between perceived and known geometry.

Once a satisfactory transformation was determined, it was added to the TF tree via a launch file or static broadcaster node, enabling consistent coordinate transformations in real-time during runtime using TF2. This allowed other ROS nodes (e.g., perception, motion planning) to query transforms between the camera and robot frames reliably using the TF2 buffer.

4.2 Transformations

The grasp obtained in AnyGrasp was now converted to world frame or robot base frame. This was done using the TF Tree. We chained the matrix operations

by first considering the transformation between camera and end effector link obtained by hand eye calibration and then the transformation between end effector and robot base frame(TF-Tree). After doing the matrix multiplications the grasp obtained was in the world frame. The transformation are achieved through a composition of two known transformations:

1. The rigid transformation from the camera to the end-effector, T_{ee}^{camera}, obtained through hand-eye calibration.
2. The current transformation of the end-effector in the world frame, T_{world}^{ee}, obtained using the robot's forward kinematics.

The resulting transformation from camera frame to world frame is given by:

$$T_{world}^{camera} = T_{world}^{ee} \cdot T_{ee}^{camera}$$

4.3 Planning

Once the grasp pose for the orange box is estimated and transformed into the robot's base frame, the next step is motion planning — computing a collision-free trajectory for the robot to move from its current state to the target pose. This is a critical step in autonomous manipulation, as it ensures that the robot can reach the object without colliding with obstacles or violating its kinematic constraints.

We formulated the planning problem as follows:

- **Start state:** The current configuration of the xArm7 robot, represented as a 7-DoF joint vector.
- **Goal pose:** The desired end-effector pose (position + orientation) for grasping the orange box, obtained from AnyGrasp and transformed into the robot's coordinate frame.
- **Obstacles:** A combination of environment geometry (walls, table) and randomly spawned 3D objects in the PyBullet simulation. These were added as collision bodies in the scene.

For planning, we used the **RRT-Connect** provided by the `pybullet-planning` library. RRT (Rapidly-Exploring Random Tree) is a sampling-based planning method particularly suited for high-dimensional, non-convex spaces like those of robotic manipulators. It works by incrementally building a tree from the start configuration by randomly sampling configurations and connecting them to the nearest node in the tree, while checking for collision-free connections.

RRT-Connect, a bi-directional variant, grows two trees — one from the start and one from the goal — and attempts to connect them. This increases convergence speed and planning success in complex environments.

Key implementation steps included:

1. We used a `collision_checker` function from `pybullet-planning` that verifies whether a given robot configuration is in collision with any objects in the environment, including self-collisions.

2. A `sample_without_collision` function was implemented to generate valid joint configurations, biased toward the goal using probabilistic sampling.

3. We used a linear `interpolate_joint_trajectory` function to extend paths between sampled configurations, ensuring smooth motion.

4. Once a valid path was found by RRT-Connect, it was post-processed and executed on the robot using joint-space interpolation.

The planning was performed entirely in joint space, and the resulting trajectory was a series of valid, collision-free joint configurations. These configurations were directly applied to the robot in PyBullet during execution. For real robots, the same path could be sent to a trajectory controller using ROS.

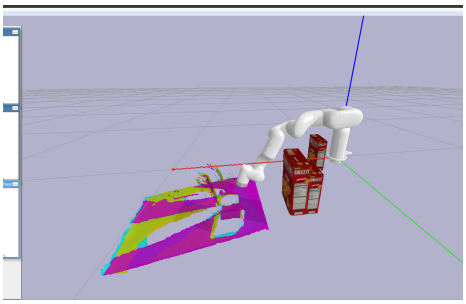

Figure 6: Final pose near the Orange box after RRT.

Advantages of RRT-Connect in our setup:

- Able to handle narrow passages and complex obstacle geometry.
- Fast convergence due to bidirectional search and goal biasing.
- Flexible to work with high-DoF arms like the xArm7 without exhaustive search.

Thus, the planning module served as the bridge between high-level perception (object detection and pose estimation) and low-level control (joint actuation), enabling the robot to autonomously grasp the object in a realistic and cluttered environment.

Pick and Place- Note that, we had implemented Pick and Place functionality in Moveit2 framework which utilized octomap and ros2. But due to a laptop issue, we could not continue on it, but we have attached the hardware demonstration links of the pick and place in the google drive.
All the video demonstrations and links of working can be found out in the doc below: Working Links