Project_Report

Starterpack, Papers and learnings

Week of June 23–29, 2025: Research and Technical Exploration

Anygrasp Model Analysis

The Anygrasp framework utilizes a two-stage architecture for robust grasp synthesis. The geometric processing module ingests point clouds and generates candidate grasp poses *G(R, t, w)*. It begins by predicting two masks via a multilayer perceptron (MLP): an objectness mask for foreground-background segmentation and a graspability heatmap identifying grippable regions. Using grasp-aware farthest point sampling, 1,024 seed points are selected from high-probability regions. For each point, 300 views are evaluated and scored by an MLP, with the optimal view selected through point view scoring (PVS). Local features are aggregated within a cylinder space —ensuring pose invariance—around each point. A modified GSNet then predicts 12 rotations and 5 approach depths per point, yielding 60 grasp scores and widths, along with 12 stability scores (one per rotation). The final grasp score combines quality and stability:
Final Score = Grasp Score × (1 − Stability Score)
Top-ranked grasps are reparameterized into 7-DoF poses for execution, balancing stability with real-time performance.

The temporal association module constructs a 256-dimensional feature vector per grasp, combining geometric features, RGB-based texture information, and 12-D pose parameters. Correspondence across frames is computed via cosine similarity in an $M \times M$ association matrix. Training uses supervised contrastive learning to pull features of corresponding grasps (within σ = 0.1 m) closer while pushing non-matches apart. Loss functions include GSNet classification/regression losses and a contrastive loss (temperature $\tau = 0.1$) for temporal consistency. At inference, features are matched against a prior buffer to enable dynamic tracking, followed by collision detection and gripper centering.

ReKep Framework Overview

ReKep addresses manipulation tasks through a two-stage optimization process. First, it computes an optimal sub-goal end-effector pose *e_gi* by minimizing auxiliary costs *λ_sub-goal* under stage constraints *C_sub-goal*. Then, it generates a trajectory *e_t:gi* to reach this sub-goal, minimizing path costs *λ_path* while satisfying path constraints *C_path*. The system incorporates backtracking to handle constraint violations by reverting to earlier stages. Keypoints are tracked at 20 Hz using visual input, with a forward model managing short-term rigid transformations of grasped keypoints relative to the end-effector. For initialization, ReKep employs a vision pipeline combining DINOv2 features, SAM segmentation, and k-means clustering to extract semantic

keypoints. GPT-4o generates task stages and constraints expressed as spatial relations between keypoints—enabling robust 3D reasoning without absolute coordinates and simplifying SO(3) rotation handling via keypoint arithmetic.

Additional Studies

Also reviewed OmniManip and Economic Grasp frameworks. Completed the first two hours of a ROS2 tutorial, covering basic setup, node creation, and communication protocols.

Week of June 30–July 2, 2025: ROS2 Mastery and Drawer Task Implementation

Completed the remaining seven hours of the ROS2 tutorial, advancing to complex functionalities including DDS communication and Gazebo simulations. These skills are critical for sophisticated task planning and multi-robot coordination.

Initiated work on a drawer opening and closing task. Adapted the Orchard pick-and-place code by modularizing motion primitives:

- Refactored `run_pick` into `run_draw` (drawer opening)
- Refactored `run_place` into `run_close` (drawer closing)
  Each function now returns corresponding poses and governs half of the full task sequence. Implemented linear motion for smooth drawer manipulation and integrated precise handle grasp detection using horizontal, grounding, and workspace filters to ensure reliable and safe operation.

Grasp Model Training Report

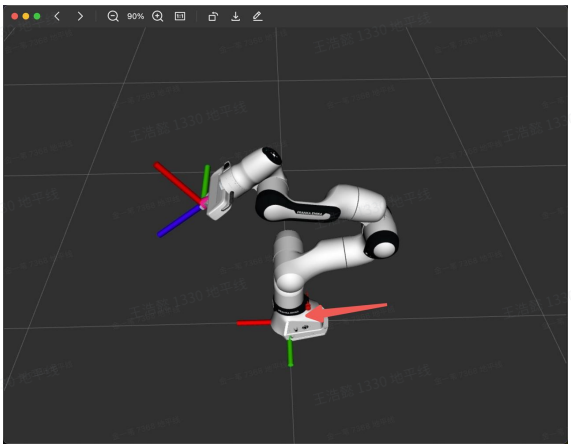
1. Introduction

This report outlines the process and findings from developing and refining a grasp model for a robotic arm, focusing on identifying and executing robust grasp poses on a drawer handle. The work was carried out in two stages: simulation-based training and real-time tuning.

---

2. Simulated Grasp Pose Development

The goal was to develop possible grasp poses for a franka robotic arm with orientation as pictured below:

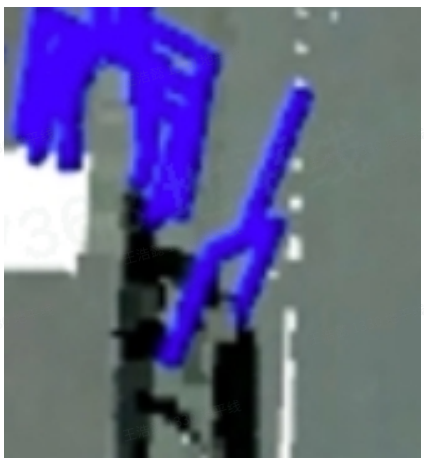


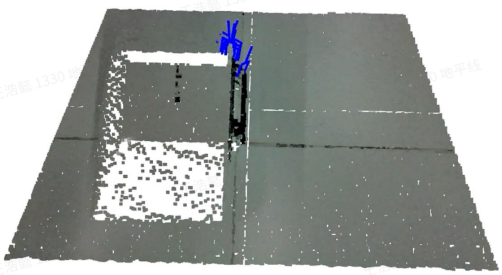
The world frame is the axis extending from the base, while the gripper had its own definition. X, Y, Z corresponds to red, green, blue respectively.

2.1 Initial Observations

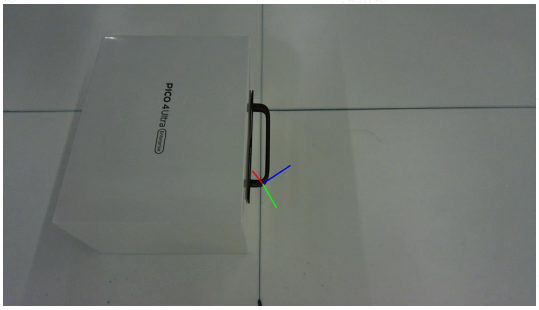
The initial grasp poses were generated in simulation using VS Code. These poses often exhibited two primary issues:

- The grasp points were frequently located near the **edges of the handle**, compromising grasp stability.





- The grasp **orientations were misaligned**, often making physical execution infeasible. As shown in the image below, the usable poses often had x-axis, the front of the arm, pointing outward, resulting in an impossible grasp.



2.2 Grounding Model Refinement

To address these problems, the grounding pipeline was redesigned into a two-stage model:

1040. **Bounding Box Grounding:** The object (box) was localized in the scene.

1041. **Handle Grounding:** A separate model or function was used to detect the handle within the bounded box region.

This approach is achieved using a double stage grounding procedure. A function takes in an object prompt and a part prompt. First, a helper gets the mask, depth, and RGB of the object and stores them to png files. Then, the part_mask, depth, and RGB are cropped from the object images. The code is visualized below:

代码块

```
        obj_rgb, obj_depth, obj_mask = self.get_object(rgb_img, depth_img, object_prompt)

        cv2.imwrite(
            os.path.join(output_folder, "object_rgb.png"),
            cv2.cvtColor(obj_rgb, cv2.COLOR_RGB2BGR)
        )
        cv2.imwrite(
            os.path.join(output_folder, "object_depth.png"),
            cv2.cvtColor(obj_depth, cv2.COLOR_RGB2BGR)
        )
        cv2.imwrite(
            os.path.join(output_folder, "object_mask.png"),
            cv2.cvtColor(obj_mask, cv2.COLOR_RGB2BGR)
        )
        part_prompts = f"<image>Please respond with segmentation mask for the {part_prompt} of {object_prompt}"  # noqa: E501
        # crop the object from the masked images
        y_indices, x_indices = np.where(obj_mask > 0)
        if len(y_indices) == 0 or len(x_indices) == 0:
            return np.zeros_like(obj_rgb), np.zeros_like(obj_depth), np.zeros_like(obj_mask), 0, 0
        x_min, x_max = np.min(x_indices), np.max(x_indices)
        y_min, y_max = np.min(y_indices), np.max(y_indices)
        cropped_rgb = obj_rgb[y_min:y_max+1, x_min:x_max+1]
        cropped_depth = obj_depth[y_min:y_max+1, x_min:x_max+1]
        input_dict = {
            "image": Image.fromarray(cropped_rgb).convert("RGB"),
            "text": part_prompts,
            "past_text": "",
            "mask_prompts": None,
            "tokenizer": self.tokenizer,
        }
        return_dict = self.model.predict_forward(**input_dict)
```

```
        masks = return_dict["prediction_masks"]
        binary_mask = np.zeros((cropped_rgb.shape[0], cropped_rgb.shape[1]),
dtype=np.uint8)
        for mask in masks:
            binary_mask = np.maximum(
                binary_mask, (mask[0] > 0).astype(np.uint8) * 255
            )
        binary_mask[binary_mask > 0] = 1
        part_masked_rgb, part_masked_depth = (
            self._apply_oriented_bbox_mask(cropped_rgb, cropped_depth,
binary_mask)
        )
        # restore the part mask to the original size
        part_mask = np.zeros_like(obj_mask)
        part_mask[y_min:y_max+1, x_min:x_max+1] = binary_mask
        part_masked_rgb_full = np.zeros_like(obj_rgb)
        part_masked_depth_full = np.zeros_like(obj_depth)
        part_masked_rgb_full[y_min:y_max+1, x_min:x_max+1] = part_masked_rgb
        part_masked_depth_full[y_min:y_max+1, x_min:x_max+1] =
part_masked_depth
        return (
            part_masked_rgb_full.astype(np.uint8),
            part_masked_depth_full.astype(np.uint16),
            part_mask,
            x_min,
            y_max
        )
```

This approach significantly improved grasp localization on the intended target.

2.3 Coordinate Transformations

The grasp inference model (Anygrasp) produces grasp poses in its own coordinates. Thus, two functions are used to convert poses from the inference coordinate to our world coordinate. First, a function is used to extract the 4x4 homogenous matrix from grasp inference results:

代码块

```
 grasp_poses_img_homo = []
        for grasp_pose_from_model in grasp_poses_from_model:
            gg_pick_homo = np.eye(4)
            gg_pick_homo[:3, :3] = grasp_pose_from_model.rotation_matrix
            gg_pick_homo[:3, 3] = grasp_pose_from_model.translation
            grasp_poses_img_homo.append(gg_pick_homo)
        return grasp_poses_img_homo
```

Then, the poses in image coordinates are transformed into world coordinates using two matrices. The first one is based off the orientation of the Anygrasp model, transforming from Anygrasp's frame to franka's frame. The second one transforms camera coordinates to world coordinates using our camera's extrinsics.

代码块

```
grasp_poses_world_homo = []
        anygrasp2franka = [
            [0.0, 0.0, 1.0, 0.0],
            [0.0, 1.0, 0.0, 0.0],
            [-1.0, 0.0, 0.0, 0.0],
            [0.0, 0.0, 0.0, 1.0]
        ]
        for grasp_pose_img in grasp_poses_img:
            grasp_pose_world_homo = (
                np.linalg.inv(self.camera_extrinsic) @ grasp_pose_img @ anygrasp2franka
            )  # get from robotwin, which is world2cam
            grasp_poses_world_homo.append(grasp_pose_world_homo)
        return grasp_poses_world_homo
```

2.4 Pose Filtering Mechanisms

To ensure grasp feasibility:

- A **retractability filter** was implemented to reject poses that could not be retracted along the approach direction without exiting the workspace. However, this filter was later deemed unnecessary due to flexible workspace boundaries.
- A **horizontal alignment filter** was added to eliminate grasp poses with vertical or non-horizontal approach vectors. This was calculated using the dot product between the grasp forward_orientation vector and the world's up-axis. If the two are vertical enough, the pose is accepted.

代码块

```
    filtered_poses = []
        remove_indices = []
        for idx, pose in enumerate(poses):
            pose_z = pose[:3, :3][
                :, 2
            ]  # z-axis of the pose, which is the grasp direction
            world_z = np.array([0, 0, 1])
            # check if the pose's z is vertical to the world z-axis
```

```
            if abs(np.dot(pose_z, world_z)) < np.cos(np.deg2rad(max_angle_deg)):
                filtered_poses.append(pose)
            else:
                remove_indices.append(idx)
        filtered_gg_picks = gg_picks.remove(remove_indices)
        return filtered_poses, filtered_gg_picks
```

---

3. Real-Time Tuning and Testing

3.1 Setup and Visual Challenges

For real-time testing, the first part was verifying grasp poses via visualization. An Ubuntu-based system was used to operate the camera without activating the robotic arm. The target object (a box with a drawer handle) was placed in a visually cluttered environment, which introduced several problems:

- **Inaccurate grounding** due to background interference. Pictured here is one bad result where the box was completely out of picture.

- **Reduced inference accuracy**, resulting in missed detections.

These were resolved by:

- **Repositioning the camera** to reduce occlusions.
- Introducing a **white background screen**, improving contrast and enabling more accurate grounding.

3.2 Pose Quality and Mechanical Constraints

In practice, for the robot to open the drawer successfully:

- The end-effector needed to approach from a **horizontal angle**.
- The grasp needed to be **centered along the handle**.

However, language-based models used for grasp inference frequently returned:

- **Tilted poses**, deviating from the horizontal plane.
- **Edge-biased grasps**, which led to unstable or failed attempts.

To correct this:

- A **centering filter** was implemented to remove grasps located at the extremes of the handle. The function filters grasp poses based on their distance from the center of a detected grounding region (the handle). It only keeps poses that are within a specified maximum distance threshold from the center point of the handle.

代码块

```
    # Convert grounding depth to 3D points
            xmap, ymap = (
                np.arange(grounding_depth.shape[1]),
                np.arange(grounding_depth.shape[0]),
            )
            xmap, ymap = np.meshgrid(xmap, ymap)
            points_z = grounding_depth / self.scale
            points_x = (xmap - self.cx) / self.fx * points_z
            points_y = (ymap - self.cy) / self.fy * points_z

            # Filter valid points
            mask = (points_z > 0) & (points_z < 1)
            grounding_points = np.stack([points_x, points_y, points_z], axis=-1)
            grounding_points_img = grounding_points[mask].astype(np.float32)

            if grounding_points_img.size == 0:
                return [], gg_picks.remove(list(range(len(poses))))

            # Transform to world coordinates
            cam2world = np.linalg.inv(self.camera_extrinsic)
            grounding_points_world = (
                grounding_points_img @ cam2world[:3, :3].T + cam2world[:3, 3]
            )

            # Calculate center of the grounding region
            center_point = np.mean(grounding_points_world, axis=0)

            filtered_poses = []
            remove_indices = []

            for idx, pose in enumerate(poses):
                # Get pose position
                pose_position = pose[:3, 3]

                # Calculate distance to center
```

```
            distance_to_center = np.linalg.norm(pose_position - center_point)

            if distance_to_center <= max_distance_threshold:
                filtered_poses.append(pose)
            else:
                remove_indices.append(idx)

        filtered_gg_picks = gg_picks.remove(remove_indices)
        return filtered_poses, filtered_gg_picks
```

- **Manual pose correction** was applied to adjust inferred poses to more stable configurations.

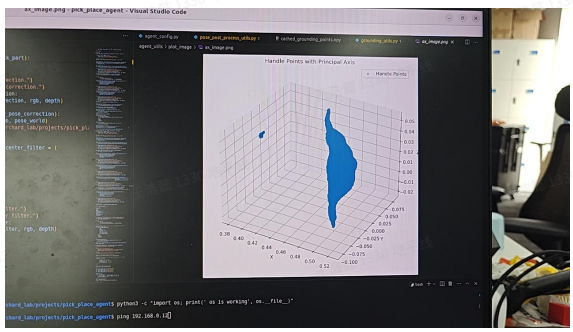
The following sections document several attempts at manual pose correction, which is straightening inferred poses so they fall within acceptable ranges for drawing the handle.

4. Limitations with PCA and Point Cloud Handling

4.1 PCA-Based Correction Attempts

Principal Component Analysis (PCA) was initially considered for automatic alignment of grasp axes, as efficiency and low computing cost was prioritized. However, this approach failed due to the characteristics of the point cloud:

- The grounded point cloud was often incomplete and heavily front-biased, particularly when the drawer was tilted. Pictured below is such an example where the entire middle part of the drawer is missing due to bad grounding results.



- PCA misaligned the primary axes due to non-uniform point distributions, resulting in inaccurate grasp orientations. Thus, PCA was deemed unsuable. An incomplete snippet of the PCA code is pictured below:

代码块

```
        center_point = np.mean(grounding_points_world, axis=0)
```

```
        centered_points = grounding_points_world - center_point

        # Perform PCA on the grounding points to find the main direction
        pca = PCA(n_components=3)
        pca.fit(centered_points)

        # Principal direction
        principal_direction = pca.components_[0] # handle direction
        perpendicular_direction_1 = pca.components_[1]
        perpendicular_direction_2 = pca.components_[2]
        principal_direction /= np.linalg.norm(principal_direction)
        world_y = np.array([0, 1, 0])
        if abs(np.dot(principal_direction, world_y)) < np.cos(np.deg2rad(20)): # tilted box
            print("box is tilted!")
            xz_points = grounding_points_world[:, [0, 2]]
            radius = 0.01  # Start with 1% of your object's size
            nbrs = NearestNeighbors(radius=radius).fit(xz_points)
            neighbor_counts = np.zeros(len(grounding_points_world), dtype=int)

        for i in range(len(grounding_points_world)):
              _, indices = nbrs.radius_neighbors([xz_points[i]])
              neighbor_counts[i] = len(indices[0])  # Number of neighbors for point i
        if abs(np.dot(principal_direction, world_y)) > np.cos(np.deg2rad(33)):
             density_threshold = np.percentile(neighbor_counts, 90)  # Keep least dense 80%
        else:
            density_threshold = np.percentile(neighbor_counts, 40)  # Keep least dense 80%
        filtered_points = grounding_points_world[neighbor_counts <= density_threshold]

        # 5. Force minimum points to remain (even if threshold fails)
        min_points_to_keep = int(0.2 * len(grounding_points_world))  # Keep at least 20%
        if len(filtered_points) < min_points_to_keep:
            filtered_points = grounding_points_world[np.argsort(neighbor_counts)[:min_points_to_keep]]

        center_point = np.mean(filtered_points, axis=0)
        centered_points = filtered_points - center_point

        # Perform PCA on the grounding points to find the main direction
        pca = PCA(n_components=3)
        pca.fit(centered_points)

```

```
        perpendicular_direction_1 = pca.components_[1]
        perpendicular_direction_2 = pca.components_[2]
        principal_direction = pca.components_[0]
```

4.2 Density-Based Filtering Approach

As an alternative, a (hopefully) **skeleton-preserving filter** was developed:

- **Dense regions** were filtered out to reduce noise.
- The **underlying structure** of the handle was retained, improving axis estimation.

This method showed **partial success**, with improved axis alignment in some cases. However, the solution remains **unstable**, particularly under varying occlusion and lighting conditions. Hardcoding of filtering thresholds also limited success. Pictured below is a snippet of the filtering code, using neighbor density to filter out dense points, which often corresponded to over-grounded regions:

代码块

```
        for i in range(len(grounding_points_world)):
            _, indices = nbrs.radius_neighbors([xz_points[i]])
            neighbor_counts[i] = len(indices[0])  # Number of neighbors for point i
        if abs(np.dot(principal_direction, world_y)) > np.cos(np.deg2rad(33)):
            density_threshold = np.percentile(neighbor_counts, 90)  # Keep least dense 80%
        else:
            density_threshold = np.percentile(neighbor_counts, 40)  # Keep least dense 80%
        filtered_points = grounding_points_world[neighbor_counts <= density_threshold]

        # 5. Force minimum points to remain (even if threshold fails)
        min_points_to_keep = int(0.2 * len(grounding_points_world))  # Keep at least 20%
        if len(filtered_points) < min_points_to_keep:
            filtered_points = grounding_points_world[np.argsort(neighbor_counts)[:min_points_to_keep]]
```

---

5. Bird-eyeview transformation approach

Seeing how malformed point clouds harmed pose estimation, a new idea was formed. The idea was to develop a fast pose correction pipeline leveraging tabletop planar geometry by utilizing

an overhead camera to capture sweeping scans of the target area. Then, using those scans to determin the world coordinates of the handle, a transformation of a previously taken complete point cloud would be made.

5.1 Implementation

- Implemented ray-plane intersection calculations with camera intrinsics
- Transformed 2D image pixels into 3D world coordinates on the desk plane
- Aimed to transform pre-captured handle point clouds into estimated real-world positions

Technical Implementation:

- Camera calibration and extrinsic parameter integration
- Plane estimation and surface normal calculations
- Ray casting and intersection point computation
- Coordinate transformation from image space to world space
- Point cloud alignment and transformation algorithms

Pictured below is the code for ray intersection and camera to world transformation:

代码块

```
fx = self.fx
fy = self.fy
cx = self.cx
cy = self.cy
extrinsics = np.array(self.camera_extrinsic)

# Extract world-to-camera components
R_world_to_cam = extrinsics[:3, :3]
t_world_to_cam = extrinsics[:3, 3]

# Convert to camera-to-world transformation
R_cam_to_world = R_world_to_cam.T  # Inverse of rotation
t_cam_to_world = -R_cam_to_world @ t_world_to_cam  # Camera position in world
t_cam_to_world[2] -= 0.16  # Apply height offset

# Step 1: Pixel to normalized camera coordinates
x_c = (u - cx) / fx
y_c = (v - cy) / fy
ray_cam = np.array([x_c, y_c, 1.0])  # Direction in camera frame

# Step 2: Transform ray to world coordinates
ray_world = R_cam_to_world @ ray_cam

```

```
    # Step 3: Intersect with ground plane (z=0)
    lambda_ = -t_cam_to_world[2] / ray_world[2]
    P_world = t_cam_to_world + lambda_ * ray_world

    return P_world[0], P_world[1]  # Return world X, Y coordinates
```

5.2 Difficulties in Implementation

There were several technical challenges in implementing. Namely: camera distortion, environmental noise, imperfection of the desk, and calibration offsets.

- Calibration Sensitivity: Accumulated errors in camera calibration parameters
- Surface Imperfections: Non-ideal tabletop flatness and subtle irregularities
- Environmental Factors: Sensitivity to lighting variations and surface reflectivity

However, there was high accuracy needed for reliable coordinate detection. Thus, results were off, as pictured below:

Here, the lines on the table represent the y and x axis. According to the world coordinates, since the handle is at the right of the x-axis, y-coordinate should be positive. However, a negative y-coordinate was returned, indicating inaccuracy.

5.3 Outcome

- Method proved ineffective for reliable grasping applications
- Accuracy insufficient for precise handle localization
- Abandoned in favor of more robust approaches
- Highlighted importance of accounting for real-world environmental imperfections

6. Prebuilt Pose Estimation Models

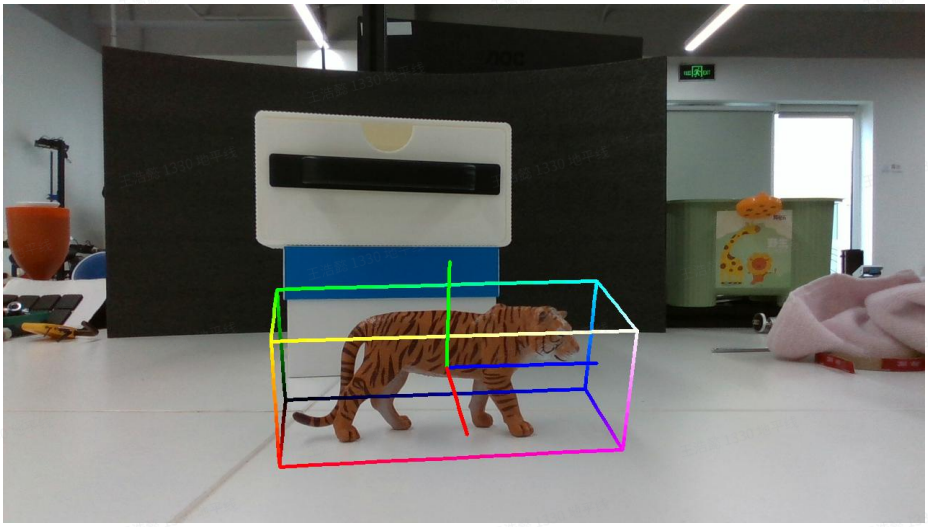
After persistent challenges with simpler local methods, I integrated sophisticated pre-trained models to leverage advanced computer vision capabilities. This strategic shift aimed to bypass

the limitations of custom solutions, which showed problems of incomplete grounding and noisy dataset.

6.1 Implementation Details

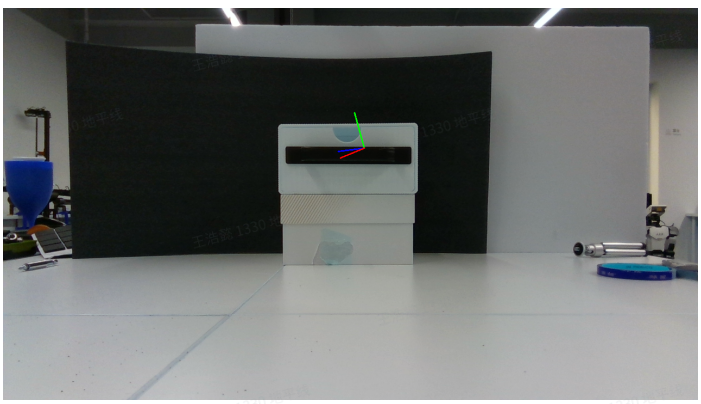
I used Omni6Dpose's GenPose2 model, which offered several advantages:

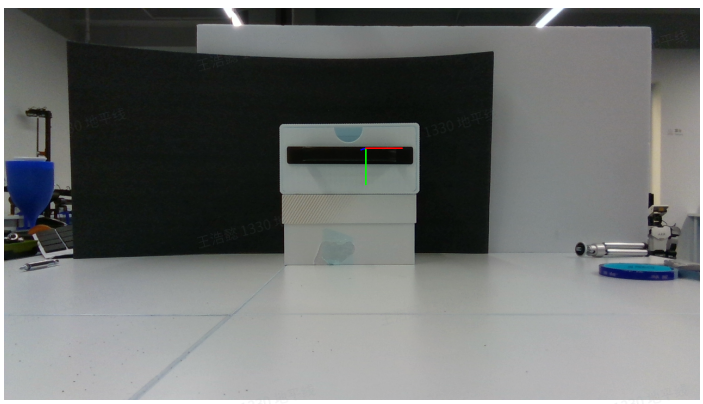
- Accurate 6D pose estimation for complex objects. Pictured below is the pose estimation for a plastic tiger toy, behind it is my drawer:

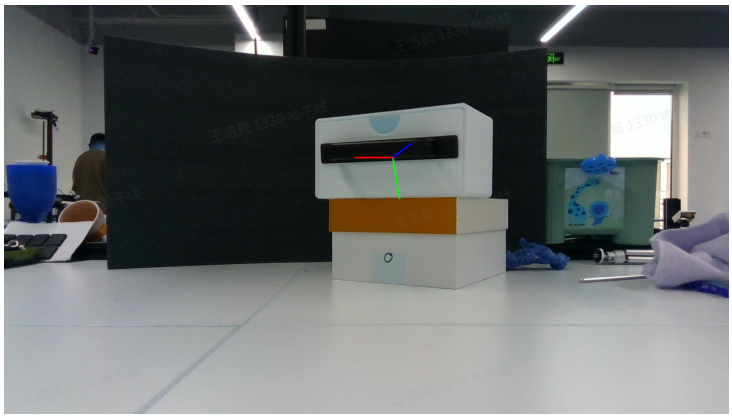


- GenPose2 offered robust handle detection in various orientations
- Real-time processing capabilities suitable for robotic applications
- Pre-trained performance requiring minimal customization

It is slower than methods such as PCA, but the delay was minimum with clever coding. As expected, the model demonstrated impressive technical capabilities by producing highly accurate pose estimations. It successfully generated well-aligned principal axes which I could use for straightening inferred poses. The quality of pose estimation significantly exceeded previous local methods, as pictured below. The left being raw inferred result, the right being straightened pose:







Above is the straightened pose for a tilted box. The pose correcting keeps the original inferred result's tilts and orientations so as to remain natural. Here is a snippet from the pose correction code built using GenPose2D. GenPose2D is used to estimate the pose of the handle, then the 4x4 homogenous matrix of the handle is extracted, and this information is used to straighten poses passed to the function. The code was messy as I was experimenting to see how GenPose2's orientation aligned with franka world's. As easily seen, without knowing the exact GenPose2Gripper transformation, this was going nowhere:

代码块

```
        rotation_matrix = grasp_poses_estimated_world[0][:3, :3]
        z_direction = rotation_matrix[:, 0]  # GenPose's X axis
        y_direction = rotation_matrix[:, 1]  # GenPose's Y axis

        corrected_poses = []

        for idx, pose in enumerate(poses):
            translation = pose[:3, 3]

            # Get direction from current position to origin
            direction_to_origin = -translation
            direction_to_origin = direction_to_origin / np.linalg.norm(direction_to_origin)

            new_z = z_direction
            new_y = y_direction

            # Ensure Z and Y are orthogonal
            if abs(np.dot(new_z, new_y)) > 1e-6:
                # Make Y orthogonal to X
                new_y = new_y - np.dot(new_y, new_z) * new_z
                new_y = new_y / np.linalg.norm(new_y)

            # Calculate initial X axis
            new_x = np.cross(new_z, new_y)
            new_x = new_x / np.linalg.norm(new_x)
```

```
26
27              # Rotate the entire coordinate system to make X point to origin ===
28              # Calculate rotation needed to align current X with
    direction_to_origin
29              rotation_axis = np.cross(new_x, direction_to_origin)
30
31              if np.linalg.norm(rotation_axis) > 1e-6:
32                  rotation_axis = rotation_axis / np.linalg.norm(rotation_axis)
33                  rotation_angle = np.arccos(np.clip(np.dot(new_x,
    direction_to_origin), -1.0, 1.0))
34
35                  # Create rotation matrix using Rodrigues' formula
36                  K = np.array([[0, -rotation_axis[2], rotation_axis[1]],
37                                [rotation_axis[2], 0, -rotation_axis[0]],
38                                [-rotation_axis[1], rotation_axis[0], 0]])
39                  R = np.eye(3) + np.sin(rotation_angle) * K + (1 -
    np.cos(rotation_angle)) * K @ K
40
41                  # Apply rotation to all axes
42                  new_z = R @ new_z
43                  new_y = R @ new_y
44                  new_x = R @ new_x  # This should now point to origin
```

6.2 Critical Limitations

However, I encountered a fundamental implementation barrier: the model's intrinsic coordinate system orientation was undocumented. This missing reference frame information prevented proper transformation of model outputs to the robot's world coordinate system.

The orientation ambiguity led to several operational issues:

- Kinematically invalid arm configurations despite accurate pose detection
- Dangerous orientation mismatches between expected and actual grasp approaches
- Unpredictable robotic movements due to undefined coordinate conventions
- Inconsistent results across different handle orientations

6.3 Resolution Efforts

I immediately engaged in multiple problem-solving approaches:

- Filed detailed GitHub issues with reproduction cases and examples
- Attempted direct communication with the model maintainers
- Conducted experimental debugging to reverse-engineer coordinate conventions
- Tested alternative transformation methodologies

However, there was no response on Github, and the experiments yielded decent but insufficient results.

6.4 Conclusion and Future Potential

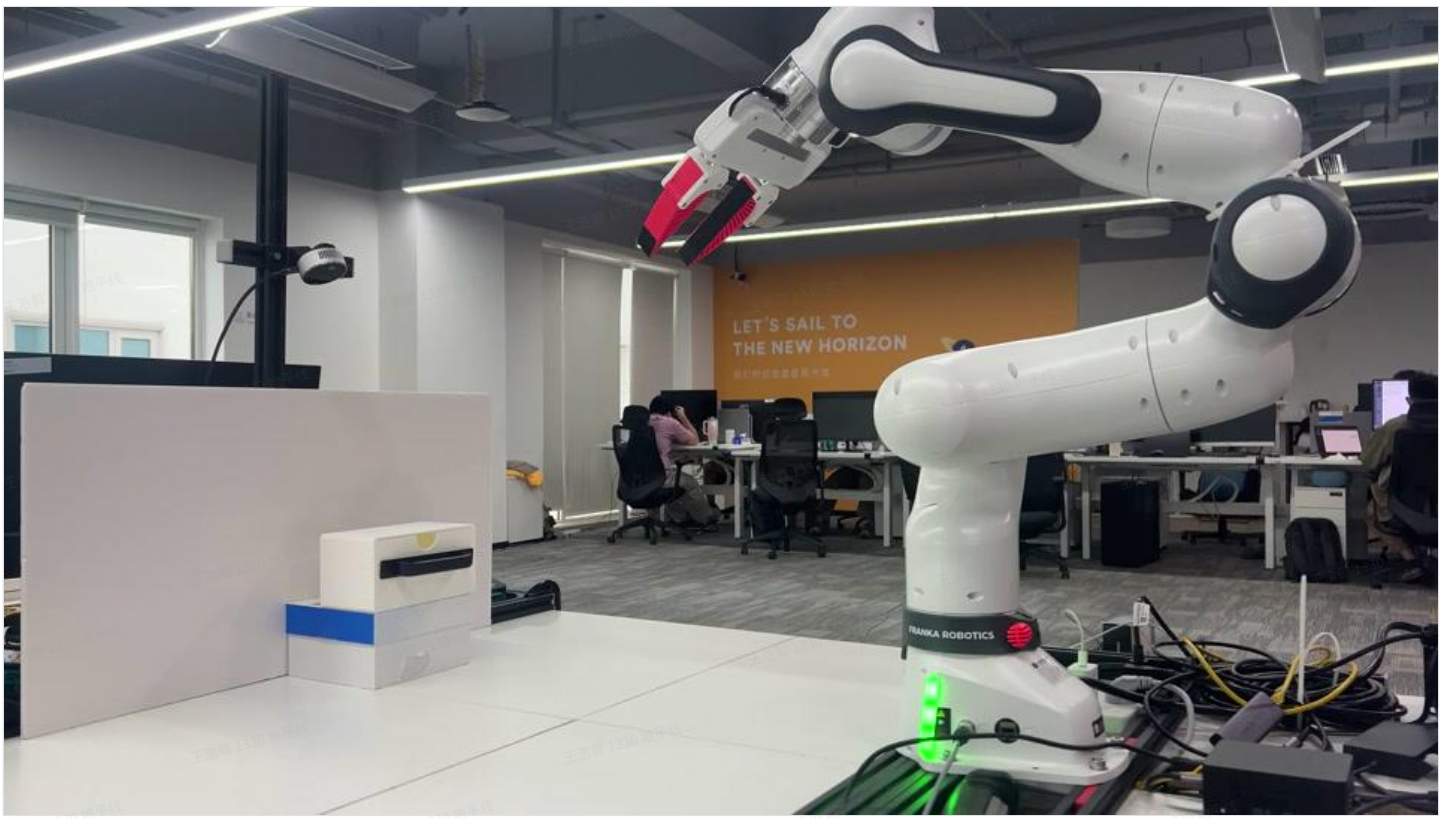
This approach represented the most promising solution despite its current limitations. The model's exceptional accuracy in ideal conditions suggested that with proper documentation or maintainer response, it could have delivered outstanding results, if not for time constraint on the internship.

7. Conclusion and Future Work

Significant progress was made in both simulation and real-time grasp inference. Key advancements included:

- A modular grounding approach that separates object and handle detection.
- Effective pose filtering strategies based on workspace constraints and approach angle.
- Practical solutions for improving grounding in cluttered real-world scenes.
- Resolution of problems generated by tilted poses generated by Anygrasp (incomplete, see Chapter 6)

Here is a video showcasing the current grasp model. As seen, the drawer was successfully opened. Gluing the drawer to the table would've showed a better drawing motion, but current approach is sufficient for demonstration:



However, challenges remain, as the current model provides decent but non-ideal grasp poses.

Pictured below is the code pipeline for run_draw_desk function:

代码块

```
def run_draw_desk(self, rgb, depth, pick_object, pick_part):
        if rgb is None or depth is None:
            logger.error("RGB or depth image is None.")
            raise ValueError("RGB or depth image is None.")

        # grasp inference
        gg_topN = self.run_grasp_inference(rgb, depth)  # noqa: N806

        # post process
        grasp_poses_img = self.pose_post_process.get_grasp_poses_img(gg_topN)
        grasp_poses_world = self.pose_post_process.grasp_poses_to_world(
            grasp_poses_img
        )
        # workspace filter
        grasp_poses_after_workspace_filter, gg_picks_after_workspace_filter = (
            self.pose_post_process.workspace_filter(grasp_poses_world, depth,
gg_topN)
        )
        if len(grasp_poses_after_workspace_filter) == 0:
            logger.error("No grasp poses after workspace filter.")
            raise ValueError("No grasp poses after workspace filter.")
        if self.config.visualize_pose_after_workspace_filter:
            self.grasp_pose_viz(gg_picks_after_workspace_filter, rgb, depth)

        part_mask, part_rgb, part_depth, x_crop, y_crop=
self.run_grounding_part(
            rgb, depth, "white drawer", "handle"
        )

        part_mask_box, part_rgb_box, part_depth_box, x_crop, y_crop=
self.run_grounding_part(
            rgb, depth, pick_object, "box"
        )
        Image.fromarray(part_rgb).save(

"/home/horizon/yiwei.jin/robo_orchard_lab/projects/pick_place_agent/agent_utils
/grounder_image/part_rgb.png"
        )
        Image.fromarray(part_depth).save(

"/home/horizon/yiwei.jin/robo_orchard_lab/projects/pick_place_agent/agent_utils
/grounder_image/part_depth.png"
        )
```

```
        Image.fromarray(part_mask).save( #* 255
"/home/horizon/yiwei.jin/robo_orchard_lab/projects/pick_place_agent/agent_utils/grounder_image/part_mask.png"
        )
        grasp_poses_after_grounding_filter, gg_picks_after_grounding_filter = (
            self.pose_post_process.grounding_filter(
                grasp_poses_after_workspace_filter,
                gg_picks_after_workspace_filter,
                part_depth,
            )
        )
        if len(grasp_poses_after_grounding_filter) == 0:
            logger.error("No grasp poses after grounding filter.")
            raise ValueError("No grasp poses after grounding filter.")
        if self.config.visualize_pose_after_grounding_filter:
            self.grasp_pose_viz(gg_picks_after_grounding_filter, rgb, depth)

        #extracting estimated pose on grounded handle
        estimated_pose = self.run_pose_estimation(
            part_rgb_box, part_depth_box, part_mask_box
        )

        estimated_list = []
        for pose in estimated_pose:
            pose = pose.reshape(4, 4)
            #print(pose_world)
            pose = pose.numpy()
            estimated_list.append(pose)

        grasp_poses_estimated_world = self.pose_post_process.grasp_poses_to_world(
            estimated_list
        )

        # horizontal filter
        grasp_poses_after_horizontal_filter, gg_picks_after_horizontal_filter = (
            self.pose_post_process.horizontal_filter(
                grasp_poses_after_grounding_filter,
                gg_picks_after_grounding_filter,
                60
            )
        )
        if len(grasp_poses_after_horizontal_filter) == 0:
            logger.error("No grasp poses after horizontal filter.")
            raise ValueError("No grasp poses after horizontal filter.")
```

```
        if self.config.visualize_pose_after_horizontal_filter:
            self.grasp_pose_viz(gg_picks_after_horizontal_filter, rgb, depth)
#gg_picks_after_horizontal_filter

        #pose correction
        grasp_poses_after_pose_correction, gg_picks_after_pose_correction = (
self.pose_post_process.pose_correction(grasp_poses_after_horizontal_filter,
                                                part_depth,
                                                grasp_poses_estimated_world,
gg_picks_after_horizontal_filter,
                                                )
        )
        if len(grasp_poses_after_pose_correction) == 0:
            logger.error("No grasp poses after pose correction.")
            raise ValueError("No grasp poses after pose correction.")
        if self.config.visualize_pose_after_pose_correction:
            self.grasp_pose_viz(gg_picks_after_pose_correction, rgb, depth)

        # center filter
        grasp_poses_after_center_filter, gg_picks_after_center_filter = (
            self.pose_post_process.center_filter(
                grasp_poses_after_pose_correction,
                gg_picks_after_pose_correction,
                part_depth,
                max_distance_threshold=0.1
            )
        )
        if len(grasp_poses_after_center_filter) == 0:
            logger.error("No grasp poses after center filter.")
            raise ValueError("No grasp poses after center filter.")
        if self.config.visualize_pose_after_center_filter:
            self.grasp_pose_viz(gg_picks_after_center_filter, rgb, depth)

        #grasp pose
        grasp_ee_poses = grasp_poses_after_center_filter

        return grasp_ee_poses
```

Future Directions

- Finish content in Chapter 6 after Github response
- Experiment with more efficient local methods for principal axes detection

Concluding words

This internship experience provided invaluable hands-on experience in applying neural networks to real-world robotics challenges. The process demonstrated the critical importance of systematic testing and validation when integrating AI components into physical systems. Each hypothesis required rigorous experimentation to confirm or refute, highlighting the iterative nature of software development.

The work at Horizon Robotics proved particularly educational for understanding the practical constraints of neural network deployment in robotic systems. The experience emphasized that successful implementation requires not just algorithmic sophistication but also robust system integration and thorough real-time validation. This comprehensive approach to testing and experimentation underscored the gap between theoretical models and operational reality.

The project provided an excellent educational experience in the complete development lifecycle of AI-enhanced robotic systems. In the future, I hope to work on more complicated projects, building off this experience.

Reference:
https://github.com/HorizonRobotics/robo_orchard_lab/tree/master/projects/pick_place_agent