COMS 4733 Homework 3

Jaisel Singh

November 13, 2024

1 Camera Calibration

1.1 Constructing the Intrinsic Matrix

The intrinsic camera matrix K encodes the internal parameters of a camera that define how 3D points in the camera coordinate frame are projected onto the 2D image plane. I implemented the `get_camera_intrinsics` function to construct this 3×3 matrix from the focal lengths (f_x, f_y) and principal point coordinates (c_x, c_y).

The intrinsic matrix has the following structure:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

where f_x and f_y represent the focal lengths in pixel units (accounting for sensor size and lens properties), and (c_x, c_y) represents the principal point—the intersection of the optical axis with the image plane. This matrix transforms 3D points in the camera frame to homogeneous 2D pixel coordinates through the projection equation $\lambda \mathbf{p} = K\mathbf{P}_c$, where $\mathbf{P}_c = [X, Y, Z]^T$ is a 3D point and $\lambda = Z$ is the depth.

Image Downsampling Question: When downsampling an image by half (from $H \times W$ to $H/2 \times W/2$ using `image[::2, ::2, :]`), the camera parameters must be adjusted to maintain equivalent imaging geometry. The new parameters should be:

$$f_x^{new} = f_x/2 \tag{2}$$
$$f_y^{new} = f_y/2 \tag{3}$$
$$c_x^{new} = c_x/2 \tag{4}$$
$$c_y^{new} = c_y/2 \tag{5}$$

All intrinsic parameters scale by the same factor as the image resolution because downsampling changes the effective pixel size. When we skip every other pixel, each new pixel represents twice the physical area, so focal lengths (measured in pixels) are halved. Similarly, the principal point's pixel coordinates are halved since the origin shifts by the same downsampling factor.

1.2 Charuco Board Detection

Camera calibration requires determining the extrinsic parameters—the rotation R and translation t that relate the camera frame to a world coordinate frame. I used OpenCV's Charuco board detection to estimate these parameters for both the front and wrist cameras.

Implementation:

1. Detected Charuco markers in each calibration image using `cv2.aruco.CharucoDetector`
2. Refined corner positions for sub-pixel accuracy
3. Computed pose using `cv2.aruco.estimatePoseCharucoBoard`, which returns rotation vector (rvec) and translation vector (tvec)
4. Converted rvec to rotation matrix using Rodrigues transformation
5. Constructed the 4×4 extrinsic transformation matrix:

$$T_{world}^{cam} = \begin{bmatrix} R & t \\ \mathbf{0}^T & 1 \end{bmatrix}$$

Results: The algorithm successfully detected markers in both camera views with sufficient coverage for reliable pose estimation. Visualization of the detected board shows coordinate axes properly aligned with the Charuco pattern, confirming accurate calibration. The computed extrinsics were saved to `data/calibration/extrinsics_result.json` for use in subsequent point cloud processing.

World Frame Orientation Question: Examining the calibration visualizations and extrinsic transformation matrices, the z-axis is parallel to the vertical direction (orthogonal to the table surface). Looking at the translation vector components in the extrinsics, the camera center's position along the z-axis is **positive**, indicating the camera is positioned above the table/calibration board. This makes physical sense as cameras must be elevated to capture a top-down or angled view of the workspace. The red, green, and blue axes in the pose visualization confirm this coordinate system orientation.

2 Point Cloud Visualization

2.1 Inverse Projection Function

The `depth_to_camera_frame_point_cloud` function reconstructs 3D geometry from 2D depth measurements by reversing the camera projection model.

Method: Given a depth image where each pixel (u, v) has an associated depth value d, the corresponding 3D point in the camera frame is computed as:

$$x = (u - c_x) \cdot d / f_x \tag{6}$$
$$y = (v - c_y) \cdot d / f_y \tag{7}$$
$$z = d \tag{8}$$

This inverse perspective projection uses the depth value directly as the z-coordinate (distance along the optical axis), while x and y are computed by "unprojecting" the pixel coordinates through the camera intrinsics. The terms $(u - c_x)$ and $(v - c_y)$ center the pixel coordinates relative to the principal point, and division by the focal lengths converts from pixels to metric units in the camera frame. The function operates vectorially over all pixels using meshgrid operations, efficiently generating the complete point cloud in a single pass.

2.2 Camera to World Transformation Function

The `transform_camera_to_world` function performs a critical coordinate transformation, mapping points from the camera's local reference frame into a shared world coordinate system.

Approach: The extrinsic matrix $T_{world}^{cam} = [R \mid t]$ represents the transformation from world to camera coordinates. To transform in the opposite direction (camera to world), we compute its inverse:

$$T_{cam}^{world} = (T_{world}^{cam})^{-1} = \begin{bmatrix} R^T & -R^T t \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where we exploit the property that $R^{-1} = R^T$ for rotation matrices (orthogonality). Points are converted to homogeneous coordinates by appending a 1, transformed using T_{cam}^{world}, then converted back to Cartesian coordinates. This unified framework handles both rotation and translation in a single matrix multiplication.

2.3 Robot Motions and Wrist Camera Extrinsics

As the robot moves, the wrist camera (rigidly mounted to the end-effector) moves with it. The `calculate_wristcam_extrinsics` function computes updated camera extrinsics for any robot configuration.

Key Insight: The relative transformation between the end-effector and wrist camera is fixed—it never changes regardless of robot motion. This can be expressed as:

$$T_{world}^{wrist} = T_{world}^{base} \cdot T_{base}^{eef}(q) \cdot T_{eef}^{wrist}$$

where T_{eef}^{wrist} is constant and $T_{base}^{eef}(q)$ depends on robot joint configuration q.

Implementation:

1. From the calibration pose, compute the fixed relationship:

$$T_{eef}^{wrist} = (T_{base}^{eef})^{-1} \cdot (T_{world}^{base})^{-1} \cdot T_{world}^{wrist}$$

2. For any new robot configuration, apply this constant transform:

$$T_{world}^{wrist}(new) = T_{world}^{base} \cdot T_{base}^{eef}(new) \cdot T_{eef}^{wrist}$$

This allows us to track the wrist camera's pose throughout the robot's workspace without re-calibration.

2.4 Visualize

The final step merges point clouds from multiple cameras into a unified 3D reconstruction of the workspace.

Pipeline:

1. Loaded RGB-D data from front camera (fixed viewpoint) and wrist camera (robot-mounted)
2. Applied inverse projection to convert depth images to 3D points in respective camera frames
3. Transformed both point clouds to world frame using their respective extrinsics
4. Applied workspace bounding box filter to remove outliers and focus on region of interest

5. Combined filtered point clouds and their RGB colors for visualization

Observations: The merged point cloud demonstrates successful multi-view fusion. The front camera provides stable, wide-field coverage of the tabletop workspace, while the wrist camera contributes detailed close-up geometry that varies with robot motion. Together, they provide complementary perspectives that reduce occlusions and improve scene completeness. The bounding box effectively isolates the relevant workspace from background clutter.

Visual Verification of Wrist Camera Extrinsics: The correctness of the computed wrist camera extrinsics can be verified through several visual cues in the Open3D point cloud visualization:

1. **Geometric alignment:** Objects visible from both cameras (e.g., the green mug, table surface) appear as single coherent 3D structures rather than duplicated or misaligned ghosts. If extrinsics were incorrect, we would see doubled objects offset from each other.

2. **Surface continuity:** The table plane and object surfaces show smooth, continuous geometry across the transition between front-camera and wrist-camera regions, indicating proper spatial registration.

3. **No obvious distortions:** Objects maintain correct proportions and shapes without warping, stretching, or unnatural deformations that would result from misaligned reference frames.

4. **Consistent scale:** Objects appear at the correct relative sizes between viewpoints, confirming that translation components of the extrinsics are accurate.

3 Semantic Segmentation

3.1 Visualize the Segmentation of Pens

Semantic segmentation assigns object-level labels to points in the 3D scene. This section extends 2D segmentation masks into 3D by associating them with the corresponding point cloud geometry.

Implementation Pipeline:

1. **Point cloud generation:** Loaded front camera RGB-D image, applied inverse projection to obtain 3D points in camera frame, then transformed to world frame

2. **Workspace filtering:** Applied bounding box constraints to isolate the tabletop region and remove background points

3. **Mask-point correspondence:** Loaded 4 binary segmentation masks (one per pen), flattened them to 1D arrays matching the point cloud indexing, and applied identical bounding box filtering to maintain alignment

4. **Color assignment:** Replaced original RGB values with distinct colors for visualization (red, green, blue, yellow for pens 0-3 respectively)

5. **Centroid calculation:** For each pen, computed the 3D centroid as:

$$\mathbf{c}_i = \frac{1}{N_i} \sum_{j=1}^{N_i} \mathbf{p}_j$$

where N_i is the number of points belonging to pen i and $\mathbf{p}_j$ are the 3D point coordinates

Technical Challenge: The primary difficulty was maintaining precise correspondence between 2D segmentation masks and 3D points through multiple filtering operations. Since the bounding box removes certain points, the mask indices must be filtered identically to preserve alignment. This was achieved by applying the same boolean indexing operation to both the point array and flattened mask arrays.

3.2 Geometric Distance Calculation

Computing pairwise distances between object centroids provides quantitative spatial relationships useful for manipulation planning and workspace analysis.

Distance Calculation: For centroids $\mathbf{c}_i$ and $\mathbf{c}_j$, the Euclidean distance is:

$$d_{ij} = \|\mathbf{c}_i - \mathbf{c}_j\| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

Pairwise Distances:

- Pen 0 to Pen 1: 0.3540 m (35.40 cm)
- Pen 0 to Pen 2: 0.2161 m (21.61 cm)
- Pen 0 to Pen 3: 0.2022 m (20.22 cm)
- Pen 1 to Pen 2: 0.1953 m (19.53 cm)
- Pen 1 to Pen 3: 0.1527 m (15.27 cm)
- Pen 2 to Pen 3: 0.1025 m (10.25 cm)

Analysis:

- **Closest pair:** Pen 2 and Pen 3 at 0.1025 m (10.25 cm)
- **Farthest pair:** Pen 0 and Pen 1 at 0.3540 m (35.40 cm)

Interpretation: These geometric relationships provide actionable information for robotic manipulation. The closest pair (pens 2 and 3) are separated by just over 10 cm, indicating objects that may require careful grasp planning to avoid collisions during pick operations. The farthest pair (pens 0 and 1) defines the maximum workspace extent at approximately 35 cm. For multi-object manipulation tasks, these distances inform collision avoidance strategies, optimal pick sequences to minimize end-effector travel distance, and potential simultaneous grasping opportunities for nearby objects.

4 Semantic Features

Self-supervised vision models like DINOv2 learn rich semantic representations without explicit labels. This section explores these learned features by computing cosine similarity between a query point's feature vector and all other pixels in the image.

Method: DINOv2 extracts a 384-dimensional feature vector for each patch in the image. For a query point at pixel (x, y), cosine similarity with all other pixels is computed as:

$$\text{sim}(q, p_i) = \frac{f_q \cdot f_{p_i}}{\|f_q\|\|f_{p_i}\|} = \frac{f_q}{\|f_q\|} \cdot \frac{f_{p_i}}{\|f_{p_i}\|}$$

where f_q is the query feature and f_{p_i} are features of all other pixels. The resulting similarity map is visualized as a heatmap where warm colors (red/yellow) indicate high semantic similarity and cool colors (blue) indicate low similarity.

4.1 Query Point Analysis

For this analysis, I selected three query points on different objects and regions to explore what semantic features DINOv2 has learned.

Query Point 1 (470, 268) - Red Pen: This point was selected on the red pen in the lower portion of the scene. The heatmap reveals strong activation (red/yellow/green regions) primarily on the pens in the scene, with the highest similarity on the queried red pen itself and moderate-to-high similarity on the other pens regardless of their color. Notably, other cylindrical objects like the green mug show moderate similarity, suggesting the model recognizes geometric shape patterns. The background table surface, walls, and robot arm display minimal activation (blue regions), demonstrating effective foreground-background segmentation.

Semantic Interpretation: DINOv2 has learned to group objects by both shape and category. The model identifies pen-like cylindrical objects and associates them together despite color differences, indicating it captures geometric and functional properties rather than purely appearance-based features. The cross-object similarity between different colored pens demonstrates the model's ability to recognize object categories invariant to color, which is essential for robust object recognition in manipulation tasks.

Query Point 2 (350, 400) - Green Mug: This query point was placed on the green mug. The heatmap shows the strongest activation concentrated on the mug itself, with significant similarity extending to other cylindrical and rounded objects including the pens. Interestingly, the pink cup in the background also shows moderate similarity, suggesting DINOv2 groups objects by their container-like properties and circular geometry. The flat background surfaces remain largely dissimilar (blue).

Comparison to Query Point 1: While Query Point 1 (pen) showed similarity across all pens with some activation on the mug, Query Point 2 (mug) shows its strongest response on the mug itself with moderate similarity to pens. This asymmetry reveals that DINOv2's feature space captures hierarchical relationships - the mug is recognized as cylindrical like pens, but has additional semantic properties (container, larger scale) that distinguish it as its own object category. The model appears to encode both fine-grained geometric features and higher-level semantic categories.

Query Point 3 (200, 150) - Background Wall: This point was selected on the white background wall in the upper-left region. The heatmap displays dramatically different behavior: strong activation (red/yellow) appears across the entire white wall region and extends to other flat, uniform background surfaces. All foreground objects (pens, mugs, containers, robot arm) show very low similarity (blue/dark blue), creating a stark foreground-background dichotomy.

Interpretation: The background query demonstrates that DINOv2 has learned scene structure and context segmentation. The model clearly distinguishes between manipulable foreground objects and environmental background surfaces. This is particularly valuable for robotic applications where identifying the "operating surface" versus objects-of-interest is fundamental. The strong similarity across all background regions regardless of lighting variations indicates the model has learned texture and semantic context rather than just pixel intensity, showing robustness to illumination changes.

4.2 Comprehensive Analysis

Analyzing the three query points together reveals key properties of DINOv2's learned representations:

Semantic Granularity: DINOv2 captures hierarchical semantic information across multiple levels. At the finest level, it distinguishes individual object instances (highest similarity on the

queried object itself). At the category level, it groups functionally similar objects (all pens show high mutual similarity). At the coarsest level, it separates major scene components (foreground objects versus background surfaces). This multi-scale representation enables both precise object localization and broader scene understanding, critical for manipulation planning where the robot must identify specific grasp targets while understanding workspace constraints.

Shape vs. Appearance: The model demonstrates strong shape-based semantic understanding that transcends superficial appearance. The red, green, blue, and yellow pens all exhibit high feature similarity despite dramatically different colors, while the white background wall (which shares color with some pens) shows near-zero similarity to foreground objects. Similarly, cylindrical objects (pens, mug) share geometric features in the embedding space. This indicates DINOv2 learns 3D shape priors and geometric structure rather than 2D color patterns, a property essential for robust perception under varying lighting and appearance conditions in real-world robotics.

Foreground-Background Separation: The stark contrast between Query Point 3 (background) and Query Points 1-2 (foreground objects) reveals sophisticated scene understanding. Background regions show strong mutual similarity and near-zero similarity to any foreground object, regardless of geometric or color properties. This demonstrates that DINOv2 has learned contextual and affordance-relevant features - it implicitly understands that tables, walls, and support surfaces constitute the "environment" while pens, mugs, and containers are "objects." This distinction aligns with manipulation-relevant affordances where some regions are graspable targets and others are constraints.

Cross-Category Relationships: Examining similarity patterns across object types reveals nuanced categorical relationships. Pens show moderate similarity to the mug (both cylindrical), but the mug shows stronger self-similarity, suggesting it occupies a distinct region in feature space. The pink cup shows some similarity to the green mug but not to pens, indicating that "container" properties are encoded separately from pure geometry. These graded similarities suggest DINOv2's feature space is not organized into discrete categories but rather represents continuous semantic relationships based on shared visual and functional properties.

Implications for Robotics: These semantic features provide several advantages over raw pixels for robotic perception:

1. **Generalization:** Object recognition across appearance variations (lighting, color, texture). The model's color-invariant pen recognition suggests it would identify objects under different illumination or with surface variations, reducing the need for extensive data collection across environmental conditions.

2. **Few-shot learning:** New object categories can be learned from minimal examples by leveraging learned geometric and semantic feature representations. A robot could learn to identify new pen types from a single demonstration by recognizing their similarity in DINOv2 feature space to known pens.

3. **Robustness:** Invariance to viewpoint, lighting conditions, and partial occlusion. The strong geometric priors mean objects remain recognizable even when partially occluded or viewed from novel angles, crucial for manipulation where objects are frequently occluded by the gripper or other objects.

4. **Semantic grouping:** Functionally similar objects share similar representations, enabling transfer of manipulation strategies. A grasping policy learned on one cylindrical object (like the green mug) could potentially transfer to other cylindrical objects (pens) due to their shared geometric features.

5. **Scene understanding:** Automatic foreground-background segmentation provides task-relevant scene structure without explicit supervision, enabling the robot to focus computational resources on manipulable objects while using background features for localization and collision avoidance.

The heatmaps demonstrate that DINOv2 learns meaningful, hierarchical semantic structures without supervision, making it a powerful foundation for robotic vision systems that must operate in diverse, unconstrained environments. The model's ability to balance geometric invariance, categorical grouping, and instance discrimination addresses fundamental challenges in robot perception for contact-rich manipulation tasks where both precise localization and semantic understanding are essential.

COMS 4733 Computational Aspects of Robotics -- Homework 3

In this homework, we will go over the perception workflow of a robotic system: from camera calibration, to point cloud construction, segmentation, and semantic feature queries.

What's covered

- Pinhole camera model
- Camera calibration
- 3D coordinate transformation
- RGB, depth, and robot trajectory data processing
- Point cloud visualization
- Segmentation mask processing
- DINO feature extraction and similarity calculation

Data

Data can be downloaded from Google Drive (LionMail required for viewing), and should be extracted under the same directory as the .ipynb file. The extracted data structure should look like:

- parent directory/
  - COMS4733_HW3.ipynb
  - data/
    - calibration/
    - recording/
    - vis/

Submission

Submit a `.zip` file containing the following:

- The notebook file (`.ipynb`) with completed codes and the outputs of the code blocks.
- A `.pdf` report, containing answers to the written questions. All written questions can be found in the text instructions of each problem.

Discussion policy

You may discuss high-level ideas with classmates, but your answers must be your own. Cite all external sources. List collaborators (names/UNIs) at the end of the PDF report.

Accessibility and extensions

For documented accommodations or emergencies, contact the instructor/TA prior to the deadline when possible.

0. Notebook Setup

We recommend running the code in your local environment. If you use a local environment, we recommend using miniconda or uv for python environment management. Most packages can be installed via pip.

Special requirements

- For pytorch, please refer to the official website for installation commands.
- **important**: Install `cv2` via `pip install opencv-contrib-python==4.12.0`. `opencv-contrib-python` and `opencv-python` are similar packages and share the name `cv2`. However, only `opencv-contrib-python` contains the aruco marker detection functions. If you have `opencv-python` previously installed, please uninstall first.
- `open3d==0.19.0` is recommended for better integration with jupyter notebook. When running open3d visualizations, i.e. `o3d.visualization.visualize_geometries` and `o3d.visualization.Visualizer()`, GUI display will pop up.
- If you have difficulties accessing the data, do not have a local GUI machine, or do not have enough CPU/RAM for the code, please report to the TAs.

GPU is **not required** for this assignment.

After you successfully set up your environment, run the following code to test that all packages can be imported correctly:

```
import json
import cv2
import numpy as np
import matplotlib.pyplot as plt
import open3d as o3d
import transforms3d
import torch
from torch.nn.functional import interpolate
import time
import glob
```

1. Camera Calibration (15 pts)

1.1 Constructing the Intrinsic Matrix (5 pts)

In this section, you will construct the 3x3 **camera intrinsics matrix** (K) using the given the camera parameters.

You are provided with:

- **Focal lengths** (f_x, f_y) (in pixels)
- **Principal point** coordinates (c_x, c_y)

Instructions:

1. Implement the missing code. (3 pts)
2. In the report, answer the following question: suppose we have an image of size H x W. For computational reasons, we want to downsample the image by half, i.e. if image is a numpy array with size (H, W, 3), we apply `image = image[::2, ::2, :]`, resulting in a (H/2, W/2, 3) array. Now suppose the H x W image was originally captured by a pinhole camera with parameters `fx, fy, px, py`. If we want to change the camera parameters, such that the captured image is directly the same with the downsampled H/2 x W/2 image ignoring differences within 1 pixel), how should we set the new `fx, fy, cx, cy` parameters? (2 pts)

```
def get_camera_intrinsics(fx: float, fy: float, cx: float, cy: float)
-> np.ndarray:
    # return: 3x3 camera intrinsics matrix K

    if fx < 0 or fy < 0:
        raise ValueError("fx and fy must be positive values")
    K = np.array([
    [fx,  0,  cx],
    [0,  fy,  cy],
    [0,   0,   1]], dtype=float)
    return K
```

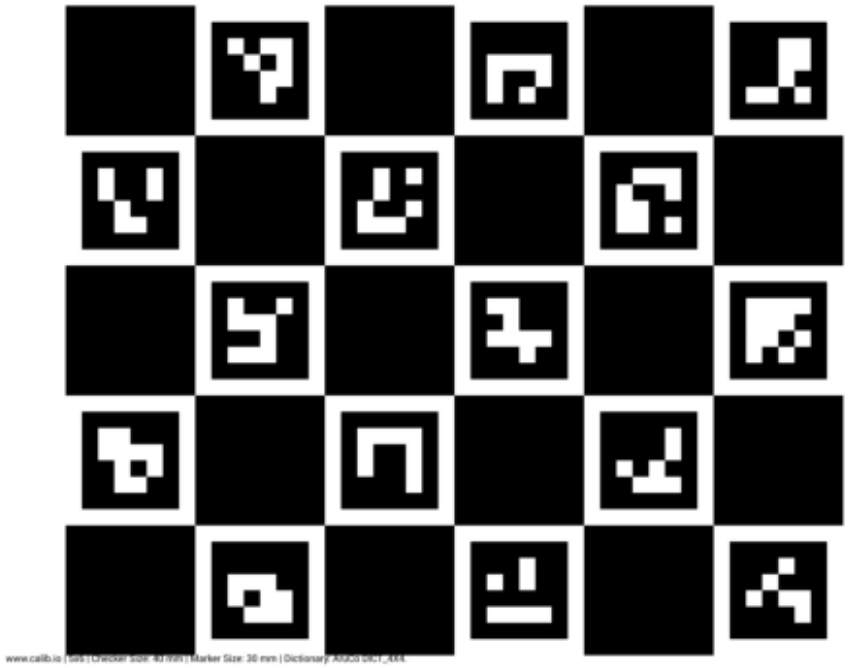
1.2 Charuco board detection (10 pts)

Below we load the 4 camera images from `data/calibration/` and run Charuco/marker detection using OpenCV. We display an overlay for each camera showing detected markers and Charuco corners.

Charuco board

We use charuco board for calibration. Charuco is a combination of Checkerboard and Aruco. A simple introduction is here. Below is a script that visualizes of the board.

```
board_image = cv2.imread('data/vis/charuco.jpg')
plt.axis('off')
plt.imshow(cv2.cvtColor(board_image, cv2.COLOR_BGR2RGB))
plt.show()
```



The detection of charuco board is done by opencv. We use its python version. The opencv-python package (can be simply used by `import cv2`) has a few predefined utilities we can use. For example, `cv2.aruco.CharucoBoard` defines the board parameters by taking in the size of the board (6x5 in our case), the edge length of the checkerboard (0.04m in our case), and the edge length of each aruco marker (0.03m in our case). Run the following code block to load the parameters and other utilities.

```
# we first define the charuco board
aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_4X4_250)
charuco_board = cv2.aruco.CharucoBoard((6, 5), 0.04, 0.03, aruco_dict)
charuco_params = cv2.aruco.CharucoParameters()
charuco_detector = cv2.aruco.CharucoDetector(charuco_board,
charuco_params)
```

Calibration function

Here we directly give the opencv function that takes in the image, detects charuco markers and visualize them, and finally returns `rvec` and `tvec`, which represents the world-to-camera rotation and translation, represented in a 3-dimensional rotation vector and 3-dimensional translation vector.

```
def calibrate(img_path, K):
    # read image
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # detect charuco board
    charuco_corners, charuco_ids, marker_corners, marker_ids =
```

```
charuco_detector.detectBoard(gray)

    vis = img.copy()
    cv2.aruco.drawDetectedMarkers(vis, marker_corners, marker_ids)
    plt.figure(figsize=(6, 4), dpi=150)
    plt.imshow(cv2.cvtColor(vis, cv2.COLOR_BGR2RGB))
    plt.title(f'Detected markers')
    plt.axis('off')
    plt.show()

    # estimate pose
    retval, rvec, tvec = cv2.aruco.estimatePoseCharucoBoard(
        charuco_corners,
        charuco_ids,
        charuco_board,
        cameraMatrix=K,
        distCoeffs=np.zeros(5, dtype=np.float32),
        rvec=None,  # placeholder
        tvec=None,  # placeholder
    )

    # Visualize pose on the image
    vis = img.copy()
    cv2.drawFrameAxes(vis, K, np.zeros(5, dtype=np.float32), rvec,
tvec, 0.1)
    plt.figure(figsize=(6, 4), dpi=150)
    plt.imshow(cv2.cvtColor(vis, cv2.COLOR_BGR2RGB))
    plt.title(f"Pose Visualization")
    plt.axis('off')
    plt.show()

    return rvec, tvec
```

Using the given functions, we can now estimate the world-to-camera transformations. Hint: `cv2.Rodrigues` is a useful function in converting 3D rotation vectors to the commonly-used 3x3 rotation matrix. Usage: `rotation_matrix, _ = cv2.Rodrigues(rvec)`

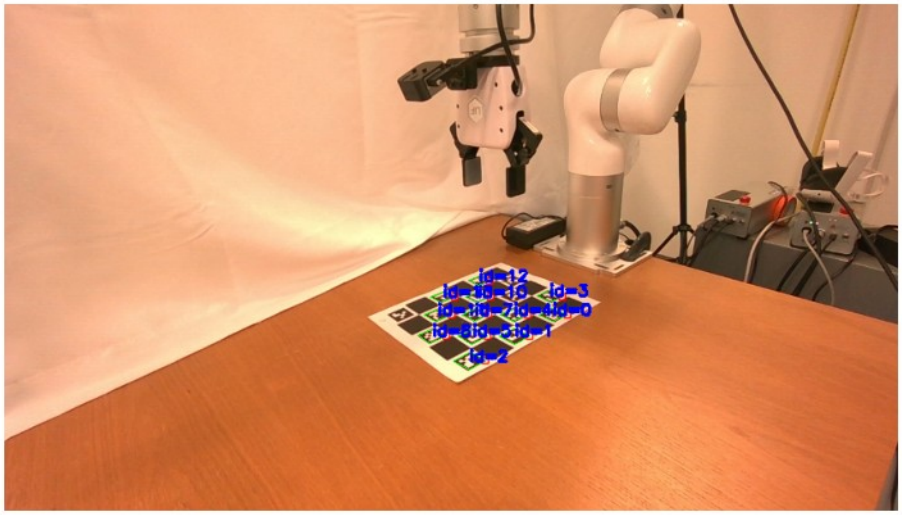
Instructions:

1. Implement the missing code. (7 pts)
2. In the report, answer the question: in the world frame, which axis is parallel to the vertical direction (orthogonal to the table surface)? On that axis, is the camera center's position positive or negative? (3 pts)

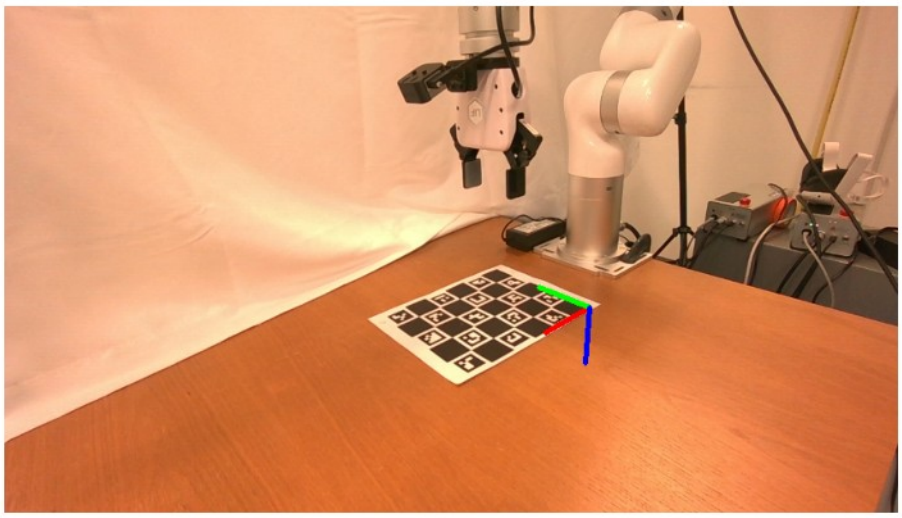
```
names = ['front', 'wrist']
img_paths = ['./data/calibration/camera_front.png',
'./data/calibration/camera_wrist.png']

# to store extrinsic matrices
extrinsics = {}
```

```
# read intrinsics file
with open('data/calibration/intrinsics.json', 'r') as f:
    intrinsics = json.load(f)

for name, img_path in zip(names, img_paths):
    # compute the calibration and get camera extrinsics

    intrinsics_dict = intrinsics[name]
    fx, fy, cx, cy = [intrinsics_dict[key] for key in ["fx", "fy",
"cx", "cy"]]
    K = get_camera_intrinsics(fx, fy, cx, cy)

    rvec, tvec = calibrate(img_path, K)
    rot_mat, _ = cv2.Rodrigues(rvec)

    world_T_camera = np.eye(4)
    world_T_camera[:3, :3] = rot_mat
    world_T_camera[:3, 3] = tvec.ravel()

    # was playing with np stacking below
    # world_T_camera = np.hstack((rot_mat, tvec))
    # world_T_camera = np.vstack((world_T_camera, [0, 0, 0, 1]))

    extrinsics[name] = world_T_camera.tolist()

# save to file
with open('data/calibration/extrinsics_result.json', 'w') as f:
    json.dump(extrinsics, f, indent=2)
```

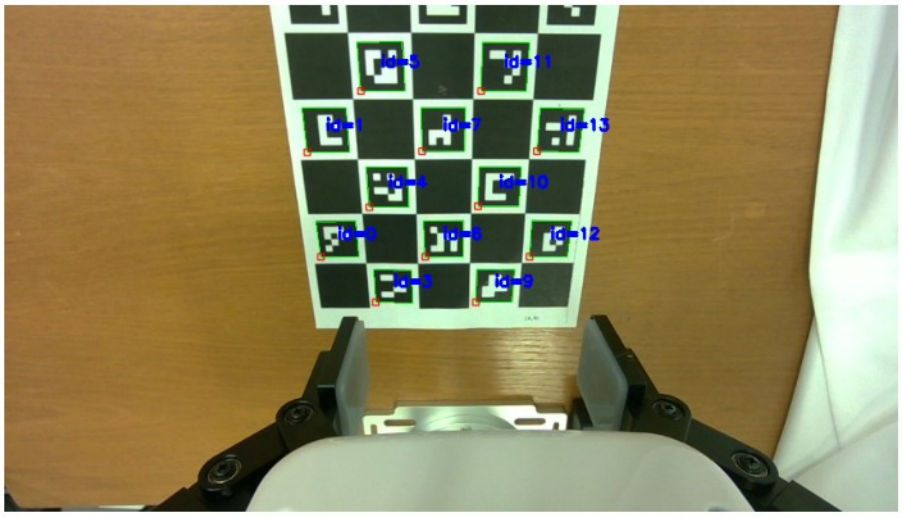
Detected markers

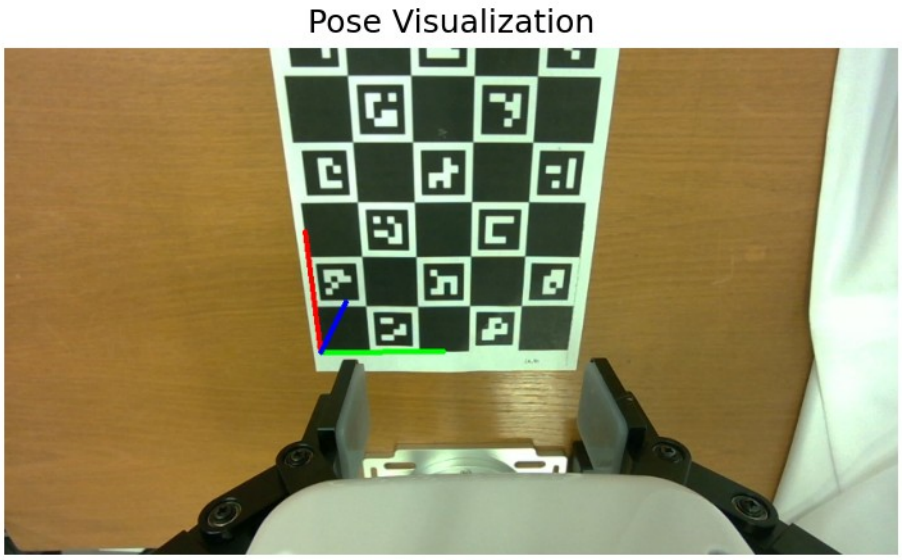


Pose Visualization



Detected markers

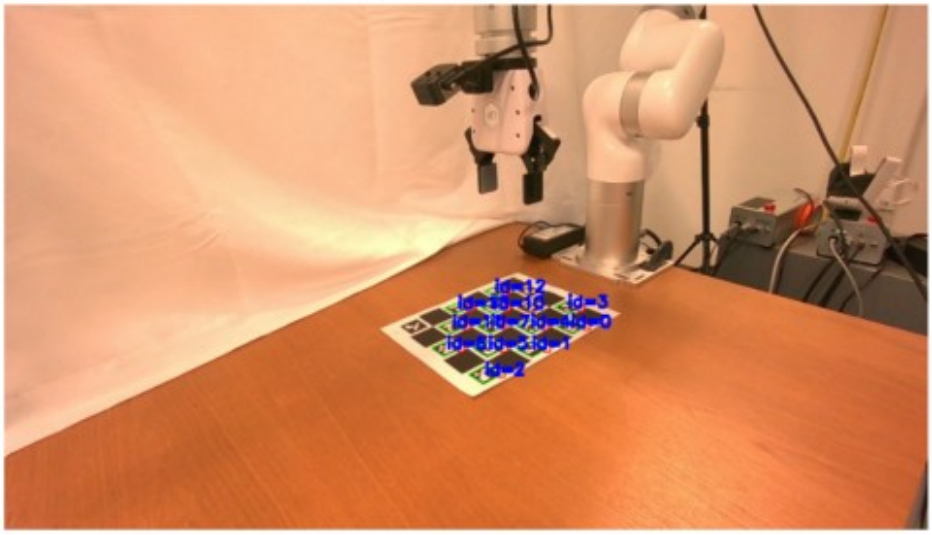




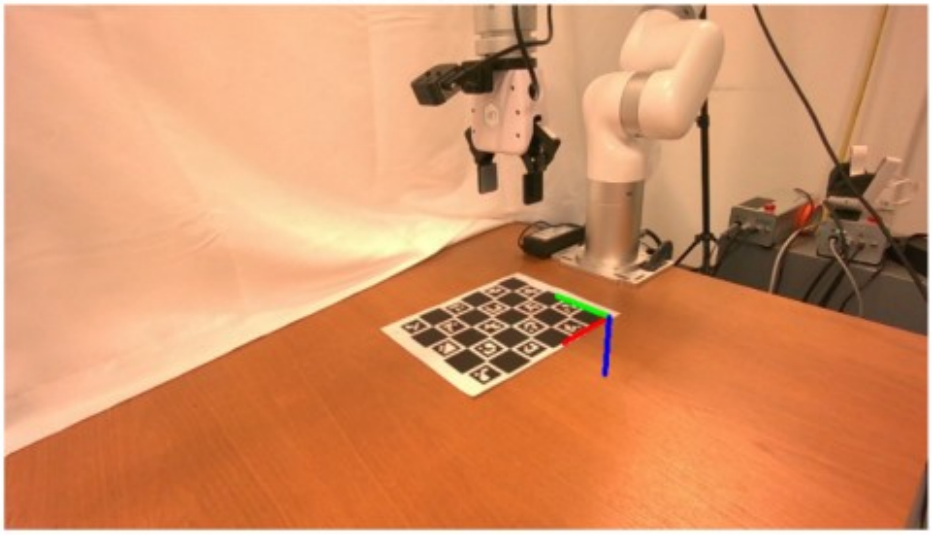
If you called the `calibrate` function correctly, you will be able to see the following visualizations (by running the script below; resolution may be different). For each camera view, the first visualization shows all the individual aruco markers detected (you can see green bounding boxes and blue texts displaying their ids); the second visualization shows the coordinate frame, located at one corner of the charuco board, and with red, green, and blue axes pointing towards the x, y, z directions of the world frame.

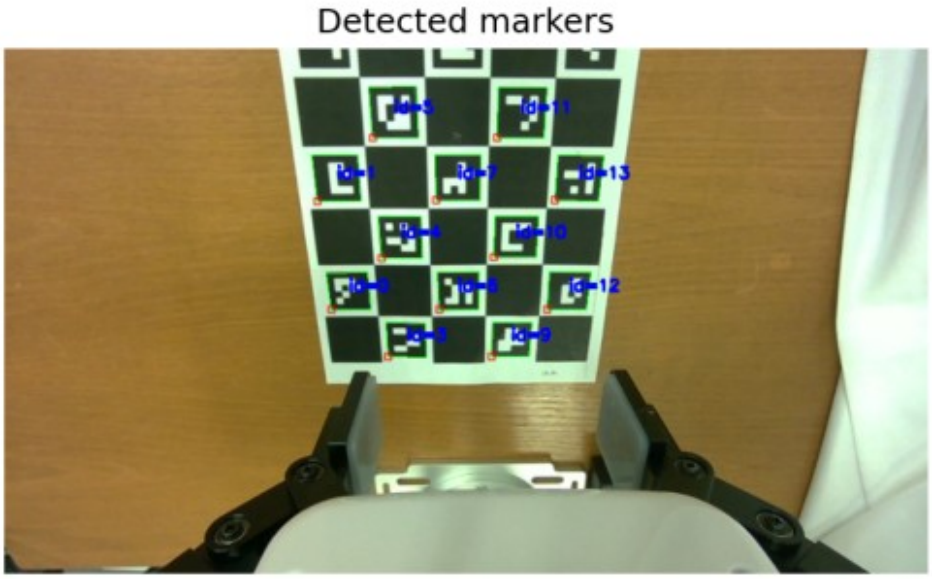
```
board_image = cv2.imread('data/vis/calibration_result_1.png')
plt.axis('off')
plt.imshow(cv2.cvtColor(board_image, cv2.COLOR_BGR2RGB))
plt.show()
board_image = cv2.imread('data/vis/calibration_result_2.png')
plt.axis('off')
plt.imshow(cv2.cvtColor(board_image, cv2.COLOR_BGR2RGB))
plt.show()
board_image = cv2.imread('data/vis/calibration_result_3.png')
plt.axis('off')
plt.imshow(cv2.cvtColor(board_image, cv2.COLOR_BGR2RGB))
plt.show()
board_image = cv2.imread('data/vis/calibration_result_4.png')
plt.axis('off')
plt.imshow(cv2.cvtColor(board_image, cv2.COLOR_BGR2RGB))
plt.show()
```

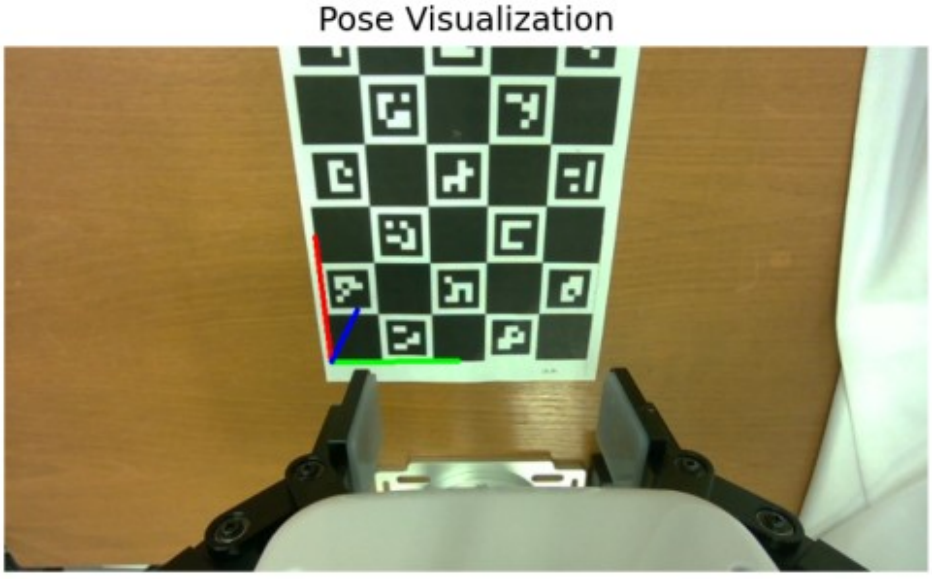
Detected markers



Pose Visualization







If you calculated correctly, the saved extrinsics file `data/calibration/extrinsics_result.json` should be identical with `data/calibration/extrinsics.json`. Note that we will check your code implementation for grading, but not the resulting file, so simply copying the file will not count.

2. Point Cloud Visualization (45 pts)

2.1 Inverse projection function (10 pts)

Given the depth image and the intrinsics, how to recover the point cloud? This is equivalent to the inverse operation of the camera space to image plane projection, which we call "inverse projection".

Instructions

- Complete the inverse projection function. You may or may not use the provided `x, y` arrays generated using the `np.meshgrid` function. (7 pts)
- In the report, write (as formulas) or briefly explain (in sentences) the mathmatical formulation of the inverse projection function. (3 pts)

```
def depth_to_camera_frame_point_cloud(depth, K):
    H, W = depth.shape
    u, v = np.meshgrid(np.arange(W), np.arange(H))

    fx = K[0, 0]
    fy = K[1, 1]
    cx = K[0, 2]
    cy = K[1, 2]

    z = depth
    # mask invalid depths
    # valid = z > 0

    x = (u - cx) * z / fx
    y = (v - cy) * z / fy

    pts = np.stack([x, y, z], axis=-1) #[valid]
    pts = pts.reshape(-1, 3) # make it N x 3
    return pts
```

2.2 Camera to world transformation function (10 pts)

After we get the point clouds in the camera frame using the function in 2.1, we want to transform the points from the camera frame to the world frame (defined by the calibration charuco board). Suppose we are given the points and the extrinsic matrix, the function needs to perform the transformation and return the points in the world frame.

Instructions

- Complete the code. (7 pts)
- In the report, write (as formulas) or briefly explain (in sentences) the mathmatical formulation of the transformation function. (3 pts)

```
def transform_camera_to_world(points, extrinsic_matrix):
    # points: Nx3 point cloud in the camera frame
```

```
    # extrinsic_matrix: 4x4 camera-to-world extrinsic matrix
    # return: Nx3 point cloud in the world frame

    # convert to the points into homogenous coordinates
    pts_hom = np.hstack([points, np.ones((points.shape[0], 1))])

    points_world_hom = (extrinsic_matrix @ pts_hom.T).T

    # convert back to be normal coordinates
    pts = points_world_hom[:, :3]

    return pts
```

2.3. Robot motions and wrist camera extrinsics (15 pts)

When we merge the point clouds from the fixed cameras and wrist cameras, one problem remains: the wrist camera is mounted to the robot. It keeps relatively static to the robot's end effector (eef), rather than the calibration board. Therefore, as the robot moves, the extrinsics (world-to-camera) matrix for the wrist camera will change. In this sub-problem, we will complete the functions for calculating the new wrist camera extrinsics.

Robot trajectores

We save the robot trajectories in the `data/recordings/robot` directory. The following function reads the robot end effector's cartesian positions and orientations relative to the robot base, and returns the transformation matrix from the robot end effector to the robot base, given a frame index. The function is directly given.

```
def read_eef_T_base(frame_idx):
    with open(f'data/recording/robot/{frame_idx:06d}.json') as f:
        data = json.load(f)
    new_ee_pos = np.array(data['obs.ee_pos'])
    new_ee_quat = np.array(data['obs.ee_quat'])
    new_ee_rot = transforms3d.quaternions.quat2mat(new_ee_quat)
    eef_T_base = np.eye(4)
    eef_T_base[:3, :3] = new_ee_rot
    eef_T_base[:3, 3] = new_ee_pos
    return eef_T_base
```

Now that we have the function to read the end effector motions, in the following function, we will calculate the wrist camera extrinsics (`world_to_new_wristcam`) given the old wrist camera extrinsics (`world_to_old_wristcam`) and the new robot pose (`new_eef_T_base`). To do this, we also need the old robot pose (`old_eef_T_base`) (i.e., the robot pose when the wrist camera calibration is done) and the robot base to calibration board transformation (`base_T_world`). These two transformations are constant, and we already hardcoded them into the function

Instructions

- Complete the function. (10 pts)

- In the report, write the mathmatical formulations for claculating the new wristcam extrinsics. (5 pts)

```
def calculate_wristcam_extrinsics(world_T_old_wristcam,
new_eef_T_base):
    # world_T_old_wristcam: 4x4 old wrist camera extrinsics
    # new_eef_T_base: 4x4 new end effector to robot base
transformation
    # return: 4x4 new wrist camera extrinsics

    # these are hardcoded constants
    old_eef_T_base = np.array([
        [1.0, 0.0, 0.0, 0.2568],
        [0.0, -1.0, 0.0, 0.0],
        [0.0, 0.0, -1.0, 0.4005],
        [0.0, 0.0, 0.0, 1.0],
    ])
    base_T_world = np.array(
        [[1.0, 0.0, 0.0, -0.185],
         [0.0, -1.0, 0.0, 0.115],
         [0.0, 0.0, -1.0, -0.01],
         [0.0, 0.0, 0.0, 1.0]]
    )
    # input: world_T_old_wristcam, new_eef_T_base

    # get world to base transform
    world_T_base = np.linalg.inv(base_T_world)

    # get world to old EE
    world_T_old_eef = world_T_base @ np.linalg.inv(old_eef_T_base)

    # use world_T_old_eef to get a relative transform between wrist
camera and eef which will stay fixed
    old_eef_T_wristcam = np.linalg.inv(world_T_old_eef) @
world_T_old_wristcam

    # get the new world to new end effector
    world_T_new_eef = world_T_base @ np.linalg.inv(new_eef_T_base)

    world_to_new_wristcam = world_T_new_eef @ old_eef_T_wristcam

    return world_to_new_wristcam
```

2.4 Visualize (10 pts)

Finally, we are ready to visualize the point clouds! We will use the recording data from the `data/recordings/` dir. The code will show the two rgb and depth images at frame 0, and launch an open3d interactive visualization of the merged point cloud. Feel free to play around with other frames as well.

Instructions

- Write the function calls to the previously defined functions. (7 pts) Note that
  - The desired output from your code block is the N x 3 `pts` array (in world frame) and `colors` array (rgb values are already normalized to 0.0-1.0, and you do not need to further modify that).
  - You might need to apply different extrinsics calculation for wrist camera (`name == 'wrist'`) and front camera (`name == 'front'`).
- In the report, include a **screenshot** of the open3d point cloud visualization, and **answer** the question: from your final results, how can we visually verify that the calculated wrist camera extrinsics is correct? (E.g., what are some visual cues in the open3d visualization?) (3 pts)

```
def construct_merged_point_cloud(frame_idx, visualize=True):
    names = ['front', 'wrist']

    # load intrinsics and extrinsics
    with open('data/calibration/intrinsics.json', 'r') as f:
        intrinsics = json.load(f)
    with open('data/calibration/extrinsics.json', 'r') as f:
        extrinsics = json.load(f)

    # Get the end-effector pose for this frame using the provided
function
    new_eef_T_base = read_eef_T_base(frame_idx)

    # to store the merged point cloud
    pts_all = []
    color_all = []

    # iterate through each camera
    for name in names:
        rgb_path =
f'./data/recording/camera_{name}/rgb/{frame_idx:06d}.jpg'
        depth_path =
f'./data/recording/camera_{name}/depth/{frame_idx:06d}.png'

        # visualize first frame
        color = cv2.imread(rgb_path)
        color = cv2.cvtColor(color, cv2.COLOR_BGR2RGB)
        color = color.astype(np.float32) / 255.0

        # visualize color image
        if visualize:
            plt.figure(figsize=(6, 4), dpi=150)
            plt.imshow(color)
            plt.axis('off')
            plt.show()

        # read depth image
```

```
        if name == 'front':
            depth = cv2.imread(depth_path, cv2.IMREAD_UNCHANGED) /
1000.0  # mm to m
        if name == 'wrist':
            depth = cv2.imread(depth_path, cv2.IMREAD_UNCHANGED) /
10000.0  # 0.1mm to m

        # visualize depth image
        if visualize:
            depth_vis = cv2.normalize(depth, None, 0, 255,
cv2.NORM_MINMAX, dtype=cv2.CV_8U)
            depth_vis = cv2.applyColorMap(depth_vis,
cv2.COLORMAP_VIRIDIS)
            depth_vis = cv2.cvtColor(depth_vis, cv2.COLOR_BGR2RGB)
            plt.figure(figsize=(6, 4), dpi=150)
            plt.imshow(depth_vis)
            plt.axis('off')
            plt.show()

        # convert depth and rgb images to point cloud (pts and colors)
        intrinsics_dict = intrinsics[name]
        fx, fy, cx, cy = [intrinsics_dict[key] for key in ["fx", "fy",
"cx", "cy"]]
        K = get_camera_intrinsics(fx, fy, cx, cy)
        camera_frame_points = depth_to_camera_frame_point_cloud(depth,
K)

        if name == 'wrist':
            world_T_old_cam = np.array(extrinsics['wrist'],
dtype=float)
            world_T_camera =
calculate_wristcam_extrinsics(world_T_old_cam, new_eef_T_base)
        else:
            world_T_camera = np.array(extrinsics[name], dtype=float)

        camera_T_world = np.linalg.inv(world_T_camera)
        pts = transform_camera_to_world(camera_frame_points,
camera_T_world)
        color = color.reshape(-1, 3)

        # filter points within bounding box for better visual quality
        bbox = np.array([[-0.5, -0.5, -1.0], [0.8, 0.5, 0.2]])
        pts_mask = np.logical_and.reduce((
            pts[:, 0] >= bbox[0, 0], pts[:, 0] <= bbox[1, 0],
            pts[:, 1] >= bbox[0, 1], pts[:, 1] <= bbox[1, 1],
            pts[:, 2] >= bbox[0, 2], pts[:, 2] <= bbox[1, 2]
        ))
        pts = pts[pts_mask]
        color = color[pts_mask]
```

```
        pts_all.append(pts)
        color_all.append(color)

    pcd = o3d.geometry.PointCloud()
    pcd.points =
o3d.utility.Vector3dVector(np.ascontiguousarray(np.concatenate(pts_all
, axis=0)).astype(np.float64))
    pcd.colors =
o3d.utility.Vector3dVector(np.ascontiguousarray(np.concatenate(color_a
ll, axis=0)).astype(np.float64))
    if visualize:
        o3d.visualization.draw_geometries([pcd])
    return pcd


_ = construct_merged_point_cloud(frame_idx=0, visualize=True)
```

Here, we also provide an interesting dynamic video visualization of the point cloud. We reconstruct the point cloud for each frame and play the resulting point cloud sequence in real time using the open3d visualizer. You can see the entire process of the robot hanging the green mug on the rack, from the front camera view. Try the code! You can also change to the wrist camera view (how?), or implement any camera motions you like. You can include interesting results in the report, but it is not required and will not be graded.

Note that the point cloud reconstruction could take a while.

The desired visualization should look like:

image

```
def video_visualization(geoms):
    vis = o3d.visualization.Visualizer()
    vis.create_window()
    vis.add_geometry(geoms[0])
```

```
    ctr = vis.get_view_control()
    params = ctr.convert_to_pinhole_camera_parameters()
    with open('data/calibration/extrinsics.json', 'r') as f:
        extrinsics = json.load(f)
    E = np.array(extrinsics['front'], dtype=float)
    params.extrinsic = E
    ctr.convert_from_pinhole_camera_parameters(params,
allow_arbitrary=True)

    vis.poll_events()
    vis.update_renderer()

    for g in geoms[1:]:
        vis.clear_geometries()
        vis.add_geometry(g)
        ctr.convert_from_pinhole_camera_parameters(params,
allow_arbitrary=True)
        vis.poll_events()
        vis.update_renderer()
        time.sleep(0.03)
    vis.destroy_window()

pts_all_list = []
for frame_idx in range(371):
    print(f"Processing frame {frame_idx}")
    pcd = construct_merged_point_cloud(frame_idx, visualize=False)
    pts_all_list.append(pcd)

video_visualization(pts_all_list)
```

```
Processing frame 0
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
last)
Cell In[11], line 29
     27 for frame_idx in range(371):
     28     print(f"Processing frame {frame_idx}")
---> 29     pcd = construct_merged_point_cloud(frame_idx,
visualize=False)
     30     pts_all_list.append(pcd)
     32 video_visualization(pts_all_list)

NameError: name 'construct_merged_point_cloud' is not defined
```

3. Semantic Segmentation (24 pts)

3.1 Visualize the Segmentation of Pens (17 pts)

We provide four masks corresponding to the pens visible in the image. Please refer to the code in Section 3.2 to: (12 pts)

1. Project the front-view RGB and depth images into a point cloud (PCD).
2. Load all four pen masks.
3. Mask the corresponding points, assign a distinct color to each pen, and visualize all of them.

For the report, please include all intermediate visualizations along with descriptions of your implementation steps and any challenges you encountered during the process. (5 pts)

The expected final visualization is shown below:

image.png

```
# Load RGB & Depth & 4 Masks
import os
name = "front"
rgb_path = './data/recording/camera_front/rgb/000000.jpg'
depth_path = './data/recording/camera_front/depth/000000.png'
mask_paths = './data/vis/masks'

color = cv2.imread(rgb_path)
color = cv2.cvtColor(color, cv2.COLOR_BGR2RGB)
color = color.astype(np.float32) / 255.0

# read depth image
depth = cv2.imread(depth_path, cv2.IMREAD_UNCHANGED) / 1000.0  # mm to
m

with open("data/calibration/intrinsics.json", 'r') as f:
    intrinsics = json.load(f)

intrinsics_dict = intrinsics[name]
fx, fy, cx, cy = [intrinsics_dict[key] for key in ['fx', 'fy', 'cx',
'cy']]
K = get_camera_intrinsics(fx, fy, cx, cy)

with open('data/calibration/extrinsics.json', 'r') as f:
    extrinsics = json.load(f)
world_T_camera = np.array(extrinsics[name], dtype=float)
camera_T_world = np.linalg.inv(world_T_camera)

camera_pts = depth_to_camera_frame_point_cloud(depth, K)
world_pts = transform_camera_to_world(camera_pts, camera_T_world)
```

```
color_flat = color.reshape(-1, 3)

# filter points inside the bounding box
bbox = np.array([[-0.5, -0.5, -1.0], [0.8, 0.5, 0.2]])

# create a boolean mask
pts_mask = np.logical_and.reduce((
    world_pts[:, 0] >= bbox[0, 0], world_pts[:, 0] <= bbox[1, 0],
    world_pts[:, 1] >= bbox[0, 1], world_pts[:, 1] <= bbox[1, 1],
    world_pts[:, 2] >= bbox[0, 2], world_pts[:, 2] <= bbox[1, 2]
))
world_pts = world_pts[pts_mask]
color_flat = color_flat[pts_mask]

mask_files = sorted([f for f in os.listdir(mask_paths) if
f.endswith(".png")])

mask_colors = [
    [1.0, 0.0, 0.0], # red
    [0.0, 1.0, 0.0], # green
    [0.0, 0.0, 1.0], # blue
    [1.0, 1.0, 0.0] # yellow
]

final_colors = color_flat.copy()

# store centroids for 3.2
centroids_3d = []
pen_names = []

for idx, mask_file in enumerate(mask_files):
    mask = cv2.imread(os.path.join(mask_paths, mask_file),
cv2.IMREAD_GRAYSCALE)
    mask_binary = (mask > 0).reshape(-1) # flatten to make it 1D

    mask_in_bbox = mask_binary[pts_mask]
    num_pen_points = np.sum(mask_in_bbox)

    # assign distinct color to this pen
    final_colors[mask_in_bbox] = mask_colors[idx]

    if num_pen_points > 0:
        # for help with 3.2
        pen_points = world_pts[mask_in_bbox]
        centroid_3d = np.mean(pen_points, axis=0)
        centroids_3d.append(centroid_3d)
        pen_names.append(mask_file.replace('.png', ''))

pcd = o3d.geometry.PointCloud()
pcd.points = o3d.utility.Vector3dVector(world_pts.astype(np.float64))
```

```
pcd.colors =
o3d.utility.Vector3dVector(final_colors.astype(np.float64))

# o3d.visualization.draw_geometries([pcd])
"""
vis_image = color.copy()  # create a copy of the original color image
mask_colors = [
    [1.0, 0.0, 0.0], # red
    [0.0, 1.0, 0.0], # green
    [0.0, 0.0, 1.0], # blue
    [1.0, 1.0, 0.0] # yellow
]

# overlay the masks onto the image
for idx, mask_file in enumerate(mask_files):
    mask = cv2.imread(os.path.join(mask_paths, mask_file),
cv2.IMREAD_GRAYSCALE)
    mask_binary = (mask > 0).astype(np.float32)

    overlay = np.zeros_like(vis_image)
    overlay[:, :, 0] = mask_binary * mask_colors[idx][0]
    overlay[:, :, 1] = mask_binary * mask_colors[idx][1]
    overlay[:, :, 2] = mask_binary * mask_colors[idx][2]

    # blend overlay onto the image
    alpha = 0.5
    vis_image = np.where(mask_binary[:, :, np.newaxis] > 0,
                         (1 - alpha) * vis_image + alpha * overlay,
                         vis_image)
    # np.where is (condition, value if true, value if false)
"""

'\nvis_image = color.copy()  # create a copy of the original color
image \nmask_colors = [\n    [1.0, 0.0, 0.0], # red \n    [0.0, 1.0,
0.0], # green \n    [0.0, 0.0, 1.0], # blue\n    [1.0, 1.0, 0.0] #
yellow\n]\n\n# overlay the masks onto the image \nfor idx, mask_file
in enumerate(mask_files):\n    mask =
cv2.imread(os.path.join(mask_paths, mask_file), cv2.IMREAD_GRAYSCALE)\
n    mask_binary = (mask > 0).astype(np.float32)\n\n    overlay =
np.zeros_like(vis_image)\n    overlay[:, :, 0] = mask_binary *
mask_colors[idx][0]\n    overlay[:, :, 1] = mask_binary *
mask_colors[idx][1]\n    overlay[:, :, 2] = mask_binary *
mask_colors[idx][2]\n\n    # blend overlay onto the image \n    alpha
= 0.5 \n    vis_image = np.where(mask_binary[:, :, np.newaxis] > 0, \n
(1 - alpha) * vis_image + alpha * overlay, \n
vis_image)\n    # np.where is (condition, value if true, value if
false)\n'
```

3.2 Geometric Distance Calculation (7 pts)

After obtaining separate point clouds for each pen, assume each pen's location is represented by the centroid of its point cloud. Please compute the closest and farthest pairwise distances among the pens. Additionally, clearly state which two pens (by color) are the closest and which two are the farthest apart. (5 pts)

For the report, please include all intermediate calculation results and provide a description of what each calculation represents. (2 pts)

```
centroids = np.array(centroids_3d)
num_pens = len(centroids_3d)

# distances matrix
min_dist = float('inf')
max_dist = 0.0
closest_pair = None
farthest_pair = None

for i in range(num_pens):
    for j in range(i+1, num_pens):
        distance = np.sqrt(np.sum((centroids[i] - centroids[j])**2))

        # Print intermediate results for report
        print(f"  {pen_names[i]} to {pen_names[j]}: {distance:.4f} m
({distance*100:.2f} cm)")

        if distance < min_dist:
            min_dist = distance
            closest_pair = (pen_names[i], pen_names[j])

        if distance > max_dist:
            max_dist = distance
            farthest_pair = (pen_names[i], pen_names[j])

print(f"Closest pair: {closest_pair[0]} and {closest_pair[1]}")
print(f"  Distance: {min_dist:.4f} m ({min_dist*100:.2f} cm)")
print(f"\nFarthest pair: {farthest_pair[0]} and {farthest_pair[1]}")
print(f"  Distance: {max_dist:.4f} m ({max_dist*100:.2f} cm)")

"""
distances = np.zeros((num_pens, num_pens))
for i in range(num_pens):
    for j in range(i+1, num_pens):
        distance = np.sqrt(np.sum((centroids[i] - centroids[j])**2))
        distances[i][j] = distance
        distances[j][i] = distance  # symmetric

# to avoid double counting pens let's just use the upper triangle
```

```
matrix
upper_tri = np.triu(distances, k=1)
unique_dist = upper_tri[upper_tri > 0]

max_dist = np.max(unique_dist)
min_dist = np.min(unique_dist)

"""
```

```
  0 to 1: 0.3540 m (35.40 cm)
  0 to 2: 0.2161 m (21.61 cm)
  0 to 3: 0.2022 m (20.22 cm)
  1 to 2: 0.1953 m (19.53 cm)
  1 to 3: 0.1527 m (15.27 cm)
  2 to 3: 0.1025 m (10.25 cm)
Closest pair: 2 and 3
  Distance: 0.1025 m (10.25 cm)

Farthest pair: 0 and 1
  Distance: 0.3540 m (35.40 cm)

"\ndistances = np.zeros((num_pens, num_pens))\nfor i in
range(num_pens):\n    for j in range(i+1, num_pens):\n        distance
= np.sqrt(np.sum((centroids[i] - centroids[j])**2))\n
distances[i][j] = distance \n        distances[j][i] = distance  #
symmetric \n\n# to avoid double counting pens let's just use the upper
triangle matrix \nupper_tri = np.triu(distances, k=1)\nunique_dist =
upper_tri[upper_tri > 0]\n\nmax_dist = np.max(unique_dist)\nmin_dist =
np.min(unique_dist)\n\n"
```

4. Semantic Features (16 pts)

In this section, you will learn how to extract DINO-V2 features from an image and write code to visualize a heatmap of feature similarity for a given query pixel. (12 pts)

In your report, please include the visualizations, explain the final heatmap, and discuss your observations and findings. Additionally, select two more query points, generate their visualizations, and provide discussions on your findings for these points as well. (4 pts)

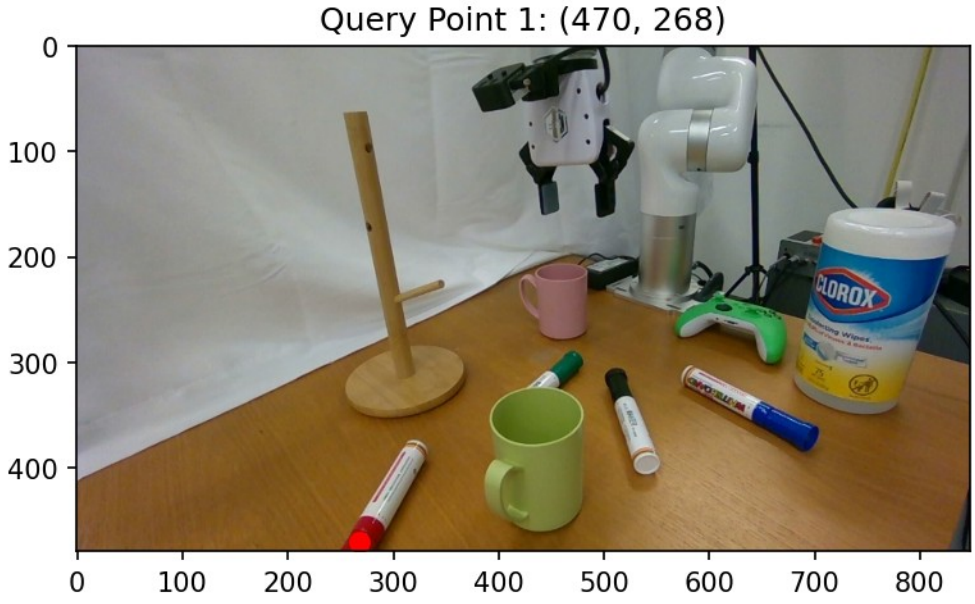
```
rgb_path = './data/recording/camera_front/rgb/000000.jpg'
rgb = cv2.imread(rgb_path)

dinov2 = torch.hub.load('facebookresearch/dinov2',
'dinov2_vits14').eval()

# Hyperparameters
patch_size = 14

# ensure input shape is compatible with dinov2
H, W, _ = rgb.shape
```

```
patch_h = int(H // patch_size)
patch_w = int(W // patch_size)
new_H = patch_h * patch_size
new_W = patch_w * patch_size
transformed_rgb = cv2.resize(rgb, (new_W, new_H))
transformed_rgb = transformed_rgb.astype(np.float32) / 255.0
img_tensors = torch.tensor(transformed_rgb,
dtype=torch.float32).permute(2, 0, 1).unsqueeze(0)

# Use DINOv2 to extract pixel-wise features
features_dict = dinov2.forward_features(img_tensors)
raw_feature_grid = features_dict['x_norm_patchtokens']  # float32
[num_cams, patch_h*patch_w, feature_dim]
raw_feature_grid = raw_feature_grid.reshape(1, patch_h, patch_w, -1)
# float32 [num_cams, patch_h, patch_w, feature_dim]

# compute per-point feature using bilinear interpolation
DINO_feature = interpolate(raw_feature_grid.permute(0, 3, 1, 2),  #
float32 [num_cams, feature_dim, patch_h, patch_w]
                                        size=(H, W),
                                        mode='bilinear').permute(0, 2,
3, 1).squeeze(0)  # float32 [H, W, feature_dim]

print(DINO_feature.shape)  # should be [H*W, feature_dim]
print(rgb.shape)

"""
# Select a point on the image
x, y = 470, 268 # Query Point
plt.figure(figsize=(6, 4), dpi=150)
plt.imshow(cv2.cvtColor(rgb, cv2.COLOR_BGR2RGB))
plt.scatter([y], [x], c='r', s=50)
plt.title(f"Query Point ({x}, {y})")

query_feature = np.array(DINO_feature[x, y, :].tolist(),
dtype=np.float32)  # float32 [feature_dim]

# Draw the heatmap figure to based on the cosine similarity
H, W, feature_dim = DINO_feature.shape
DINO_flat = DINO_feature.reshape(-1, feature_dim) # [H*W, feature_dim]

query_tensor = torch.tensor(query_feature).unsqueeze(0)
feature_tensor = torch.tensor(DINO_flat)

# normalized for computing cosine similarity
query_norm = query_tensor / query_tensor.norm(dim=1, keepdim=True)
feature_norm = feature_tensor / feature_tensor.norm(dim=1,
keepdim=True)

similarity = (query_norm @ feature_norm.T).squeeze(0)
```

```
similarity_map = similarity.reshape(H, W).numpy()
"""
H, W, feature_dim = DINO_feature.shape
DINO_flat = DINO_feature.reshape(-1, feature_dim) # [H*W, feature_dim]

query_points = [
    (470, 268, "Query Point 1"),
    (350, 400, "Query Point 2"),
    (200, 150, "Query Point 3")
]

# Draw the heatmap figure based on the cosine similarity
for x, y, label in query_points:
    plt.figure(figsize=(6, 4), dpi=150)
    plt.imshow(cv2.cvtColor(rgb, cv2.COLOR_BGR2RGB))
    plt.scatter([y], [x], c='r', s=50)
    plt.title(f"{label}: ({x}, {y})")
    plt.show()

    query_feature = np.array(DINO_feature[x, y, :].tolist(),
dtype=np.float32)
    query_tensor = torch.tensor(query_feature).unsqueeze(0)
    feature_tensor = torch.tensor(DINO_flat)

    query_norm = query_tensor / query_tensor.norm(dim=1, keepdim=True)
    feature_norm = feature_tensor / feature_tensor.norm(dim=1,
keepdim=True)

    similarity = (query_norm @ feature_norm.T).squeeze(0)
    similarity_map = similarity.reshape(H, W).numpy()

    print(f"Similarity range: [{similarity_map.min():.3f},
{similarity_map.max():.3f}]")

    fig, axes = plt.subplots(1, 2, figsize=(14, 6))

    axes[0].imshow(cv2.cvtColor(rgb, cv2.COLOR_BGR2RGB))
    axes[0].scatter([y], [x], c='red', s=100, marker='x',
linewidths=3)
    axes[0].set_title(f'Original Image\n{label}: ({x}, {y})',
fontsize=12)
    axes[0].axis('off')

    im = axes[1].imshow(similarity_map, cmap='jet', vmin=0, vmax=1)
    axes[1].scatter([y], [x], c='red', s=100, marker='x',
linewidths=3)
    axes[1].set_title('Feature Similarity Heatmap\n(Cosine
Similarity)', fontsize=12)
    axes[1].axis('off')
```

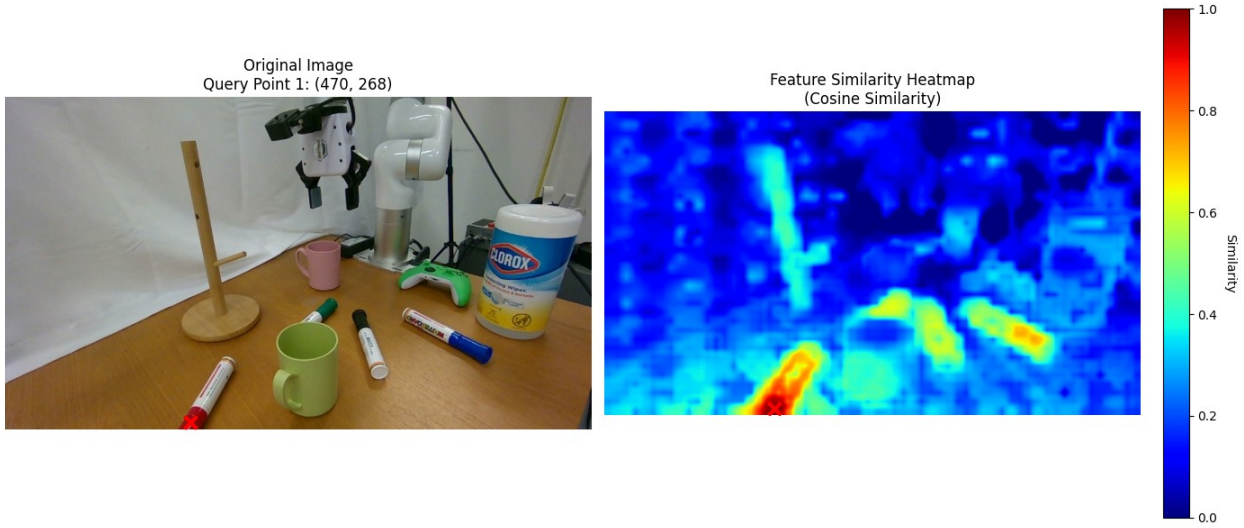
```
    cbar = plt.colorbar(im, ax=axes[1], fraction=0.046, pad=0.04)
    cbar.set_label('Similarity', rotation=270, labelpad=15)

    plt.tight_layout()
    plt.show()

    fig, ax = plt.subplots(1, 1, figsize=(10, 8))
    ax.imshow(cv2.cvtColor(rgb, cv2.COLOR_BGR2RGB))
    im = ax.imshow(similarity_map, cmap='jet', alpha=0.5, vmin=0,
vmax=1)
    ax.scatter([y], [x], c='white', s=150, marker='x', linewidths=4,
edgecolors='black')
    ax.set_title(f'Similarity Heatmap Overlay\n{label}: ({x}, {y})',
fontsize=14)
    ax.axis('off')
    cbar = plt.colorbar(im, ax=ax, fraction=0.046, pad=0.04)
    cbar.set_label('Similarity', rotation=270, labelpad=15)
    plt.tight_layout()
    plt.show()
```

```
Using cache found in
/Users/jaiselsingh/.cache/torch/hub/facebookresearch_dinov2_main

torch.Size([480, 848, 384])
(480, 848, 3)
```

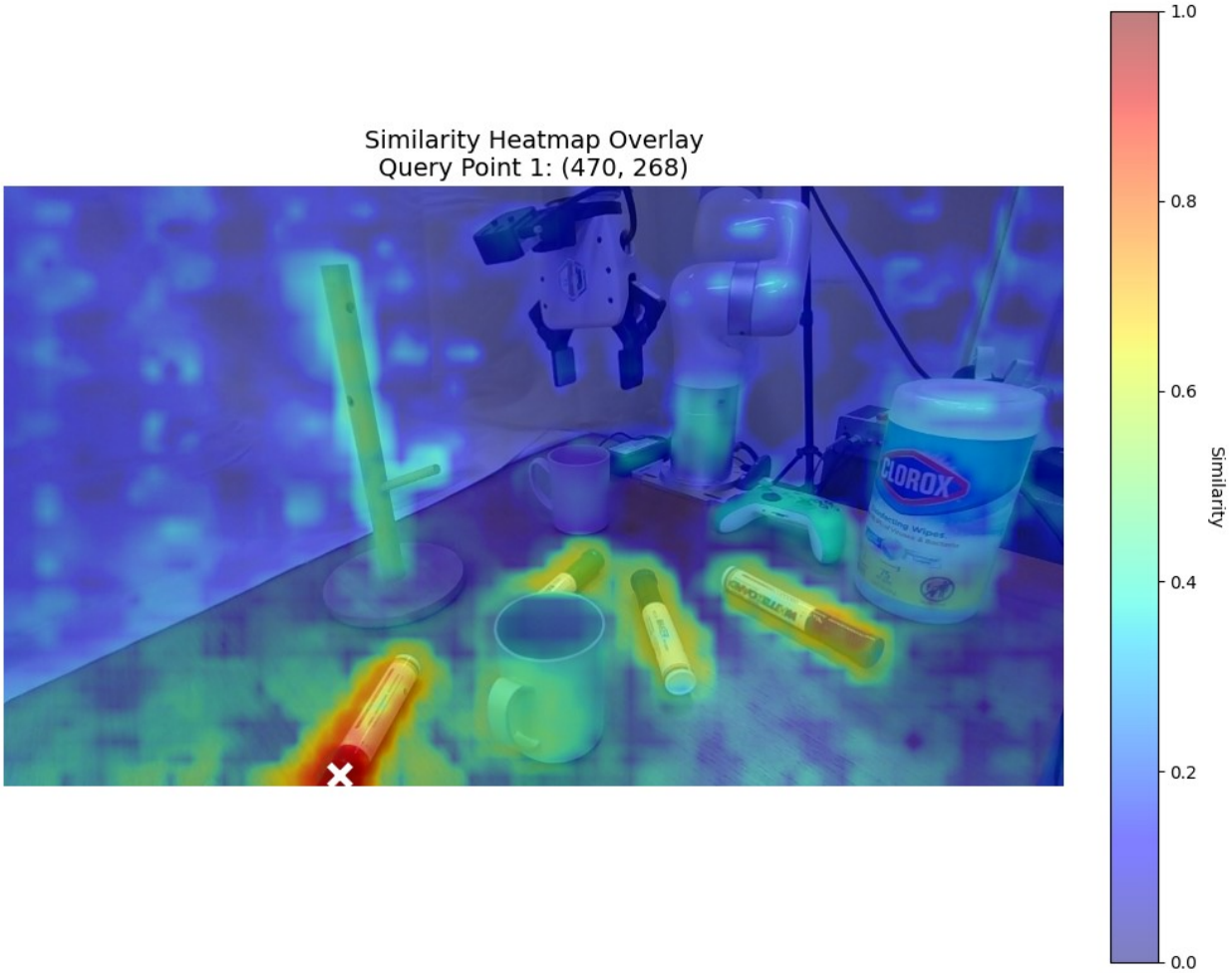


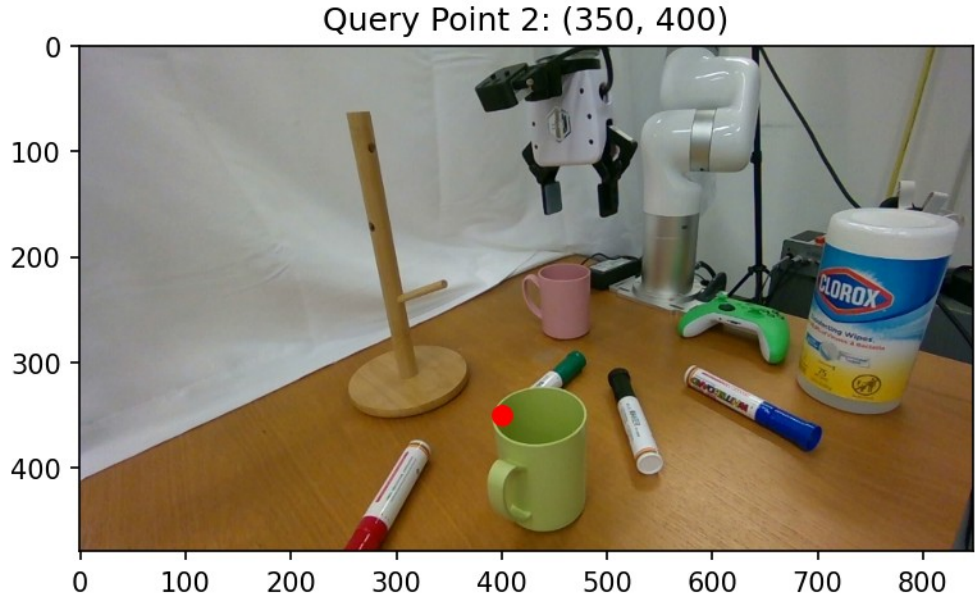

```
Similarity range: [-0.134, 1.000]
```

```
/var/folders/xk/f342x5yd6c77rjz1q6jc99jm0000gn/T/
ipykernel_17549/383732146.py:75: UserWarning: To copy construct from a
tensor, it is recommended to use sourceTensor.detach().clone() or
sourceTensor.detach().clone().requires_grad_(True), rather than
torch.tensor(sourceTensor).
  feature_tensor = torch.tensor(DINO_flat)
```

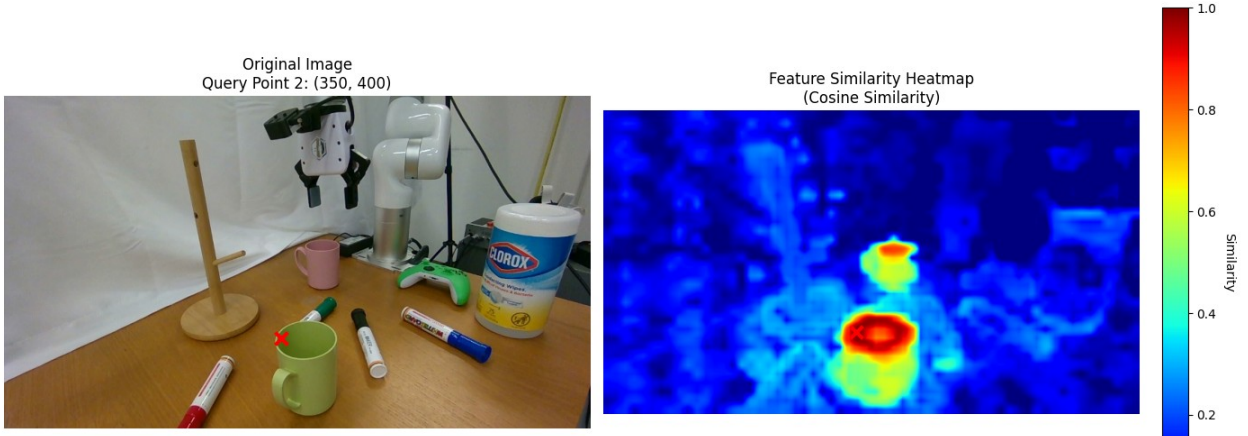


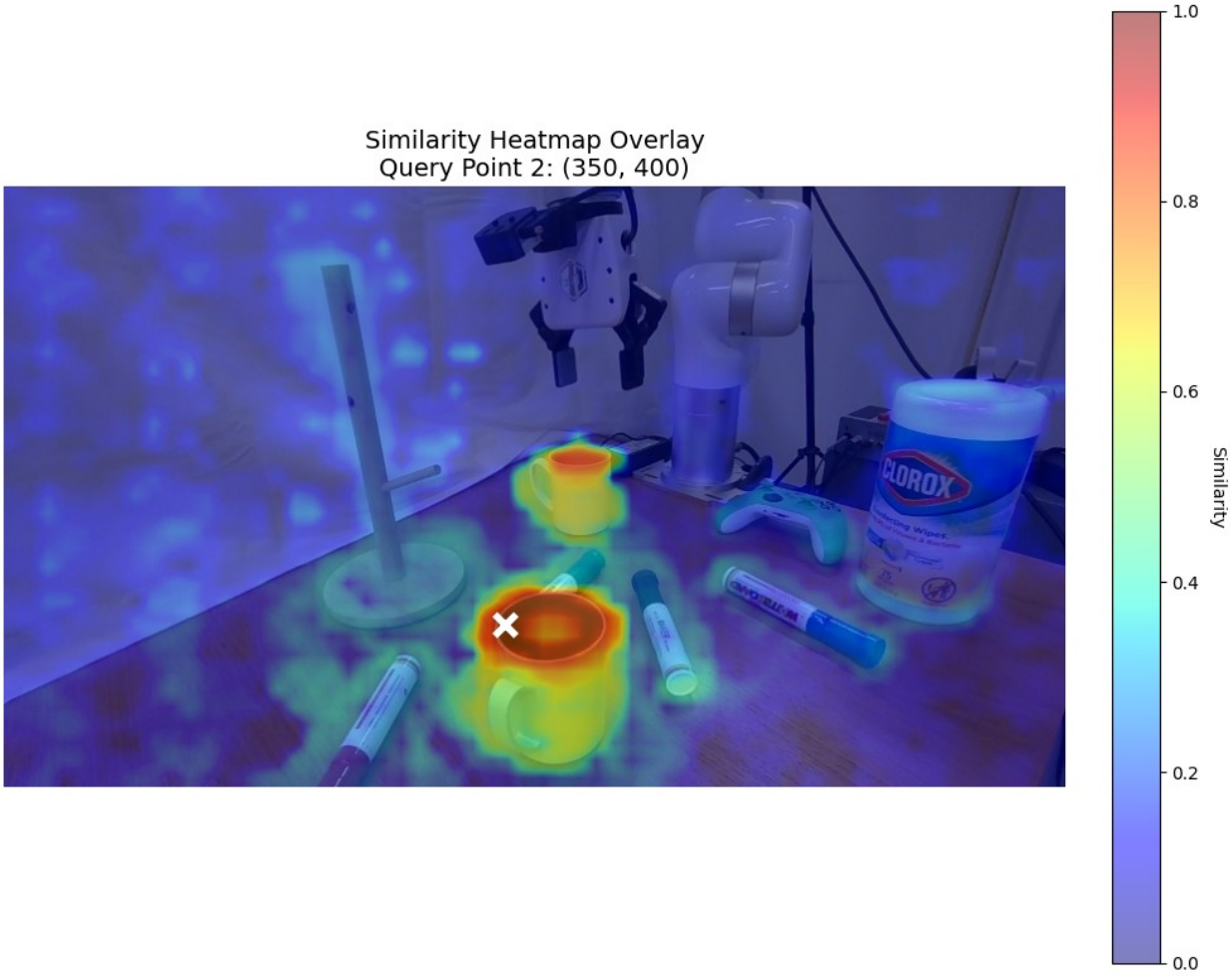

```
/var/folders/xk/f342x5yd6c77rjz1q6jc99jm0000gn/T/
ipykernel_17549/383732146.py:106: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').
Matplotlib is ignoring the edgecolor in favor of the facecolor.  This
behavior may change in the future.
  ax.scatter([y], [x], c='white', s=150, marker='x', linewidths=4,
edgecolors='black')
```

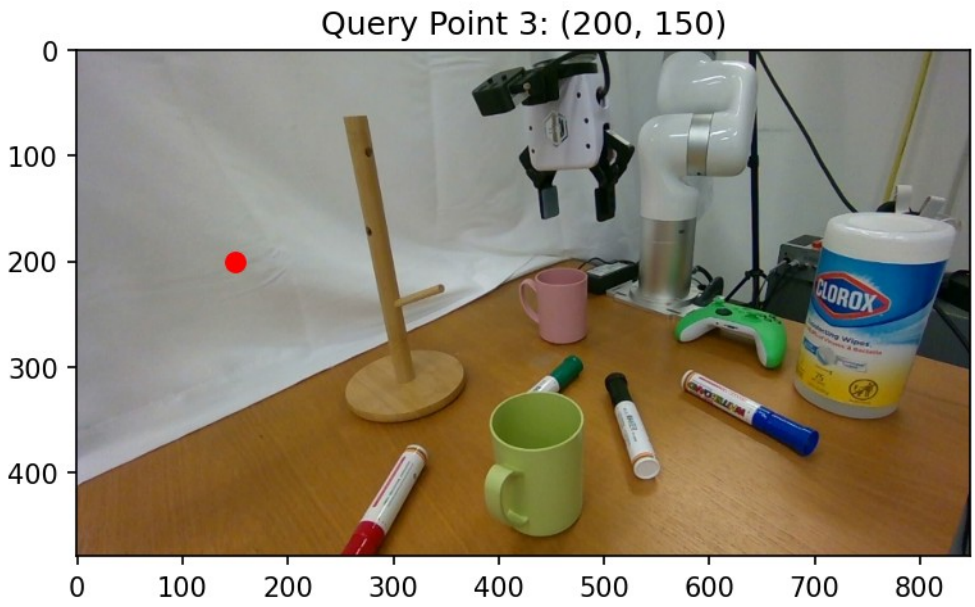

Similarity Heatmap Overlay
Query Point 1: (470, 268)
CLOROX
1.0
0.8
0.6
0.4
0.2
0.0
Similarity


Query Point 2: (350, 400)
CLOROX
0
100
200
300
400
0
100
200
300
400
500
600
700
800

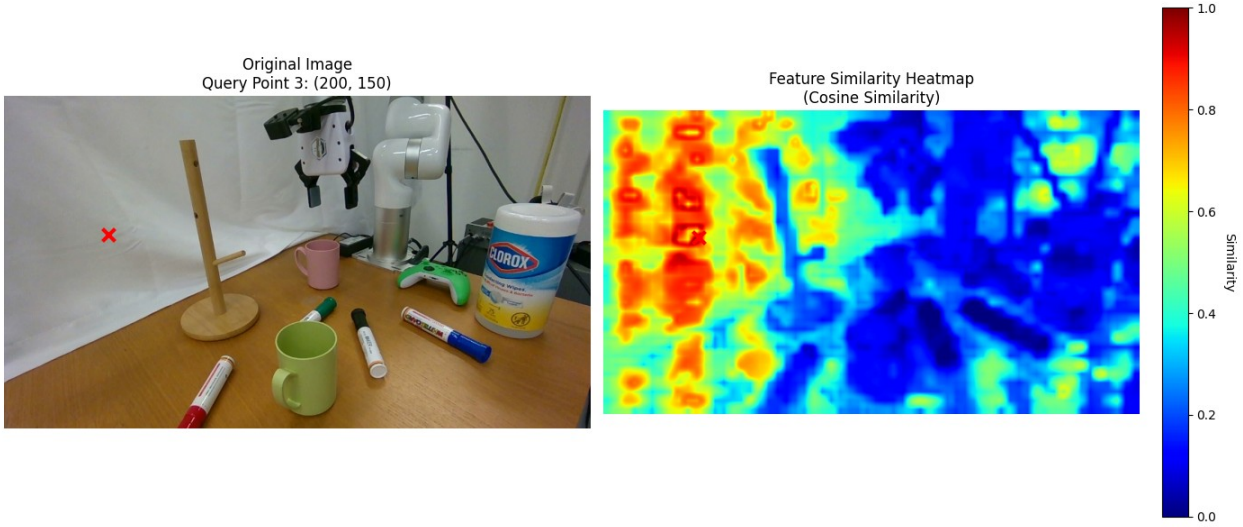
```
Similarity range: [-0.110, 1.000]
```

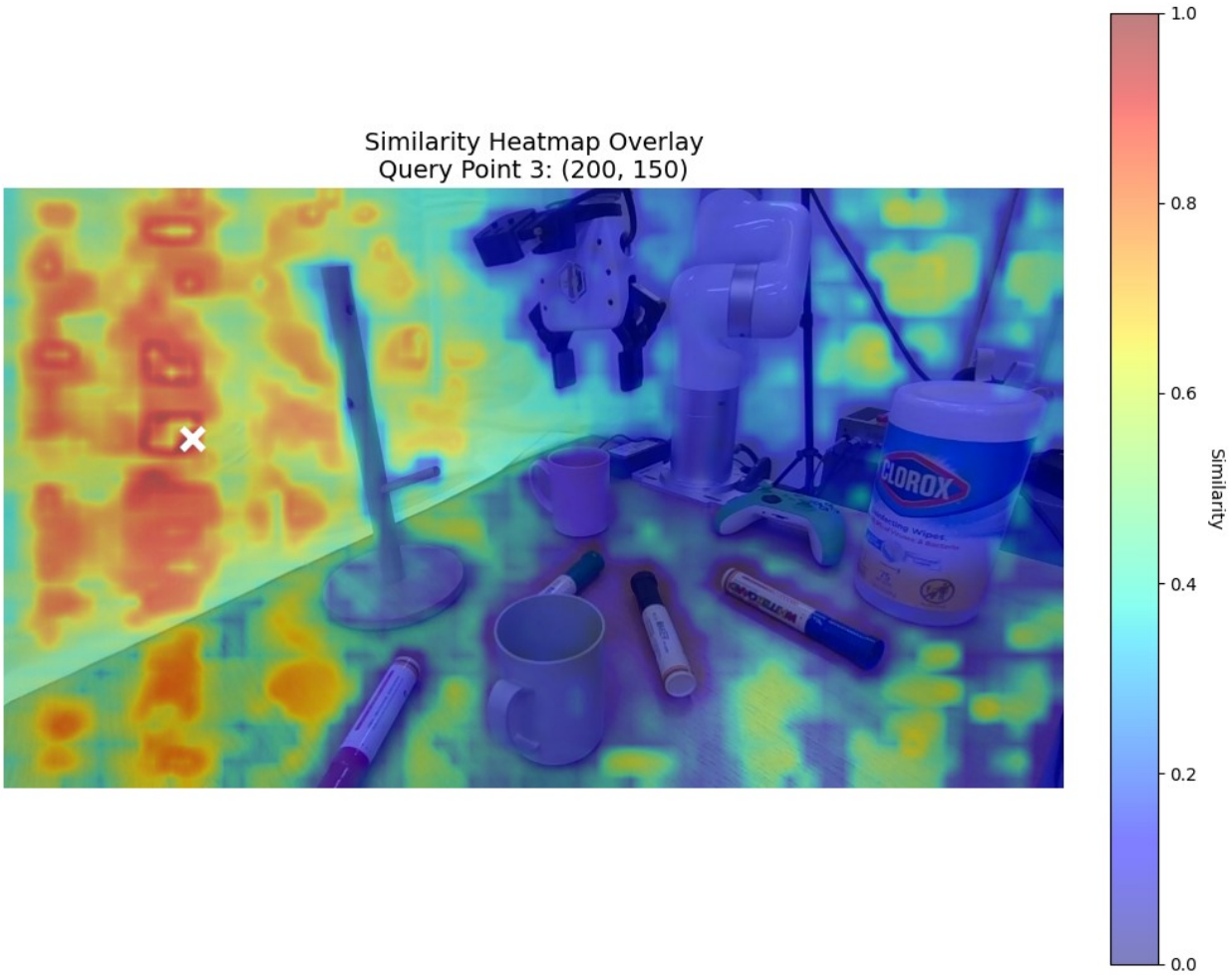







```
Similarity range: [-0.035, 1.000]
```





Submission checklist

Include all the problems in your PDF report, and keep all the outputs of the notebook. A list of the question indices are:

`1.1, 1.2`

`2.1, 2.2, 2.3, 2.4`

`3.1, 3.2`

`4`