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
```

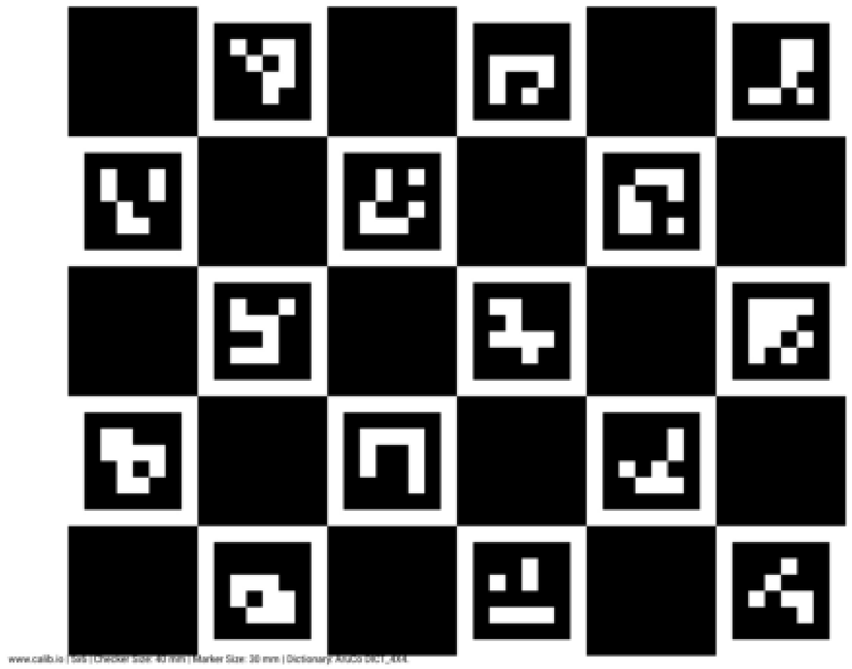
1.2 Charuco board detection (10 pts)

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()
```


www.calib.io | 5x6 | Checker Size: 40 mm | Marker Size: 30 mm | Dictionary: ArUco DICT_4X4

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)`

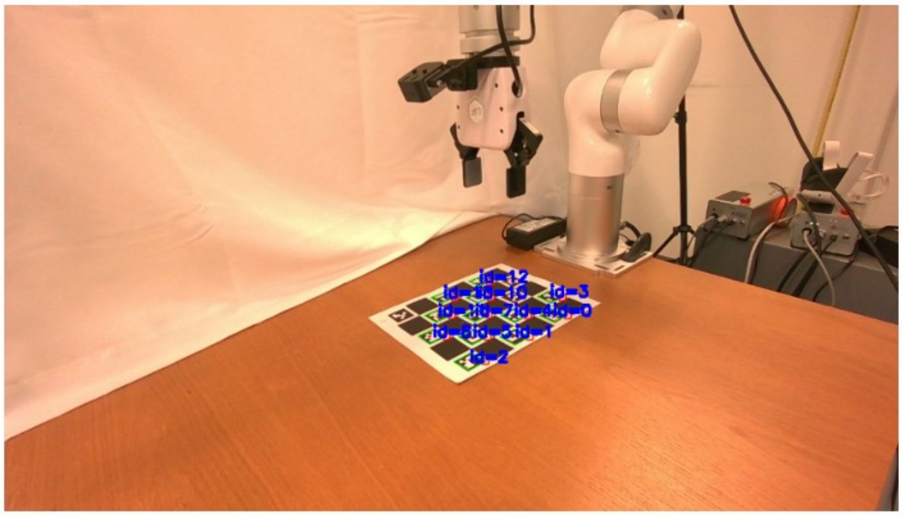
Instructions:

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)

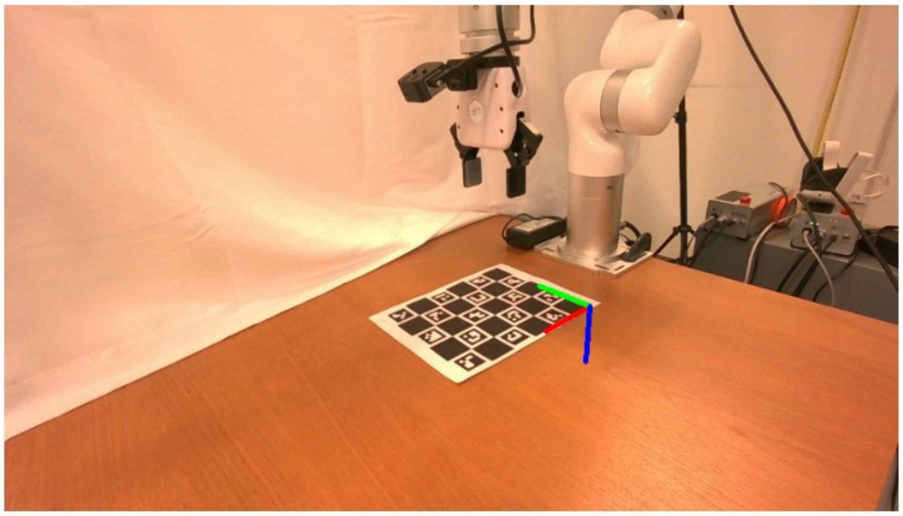
```
names = ['front', 'wrist']
img_paths = ['./data/calibration/camera_front.png',
'./data/calibration/camera_wrist.png']

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

```python
# 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)
```

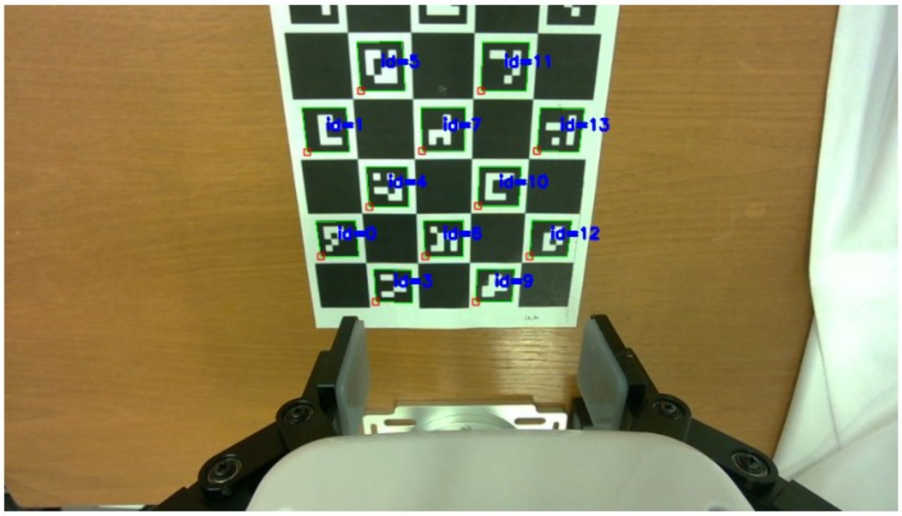
Detected markers

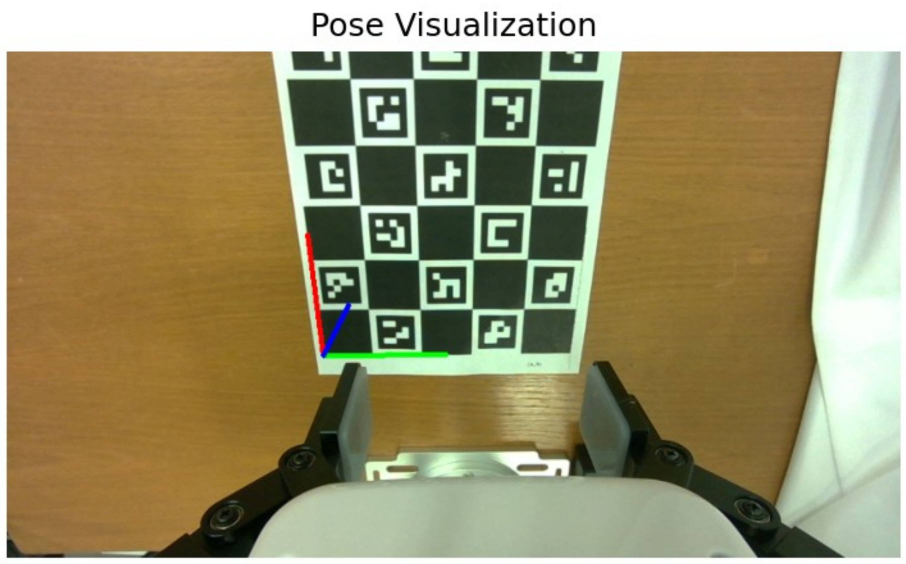


Pose Visualization



Detected markers

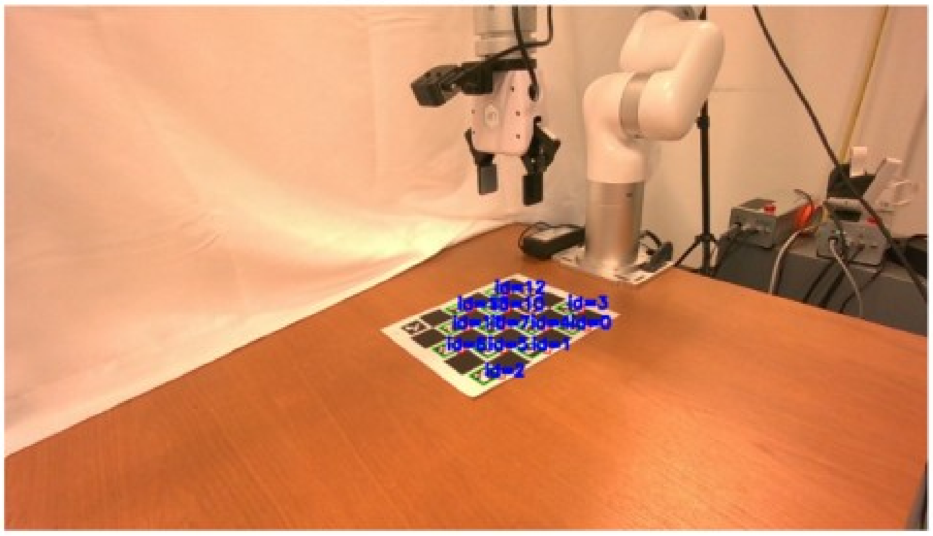




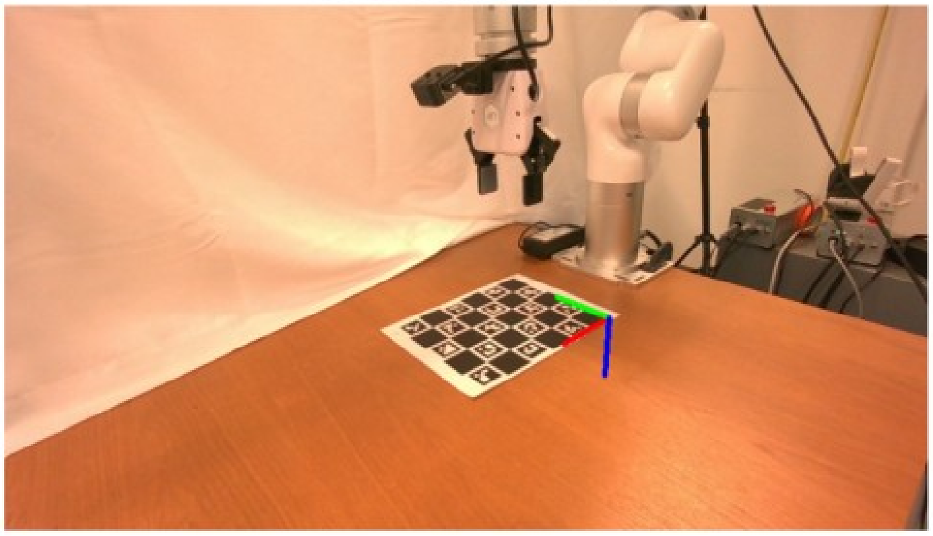
If you called the `calibrate` function correctly, you will be able to see the following visualizations (by running the script below; resolution may be different). For each camera view, the first visualization shows all the individual aruco markers detected (you can see green bounding boxes and blue texts 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()
```

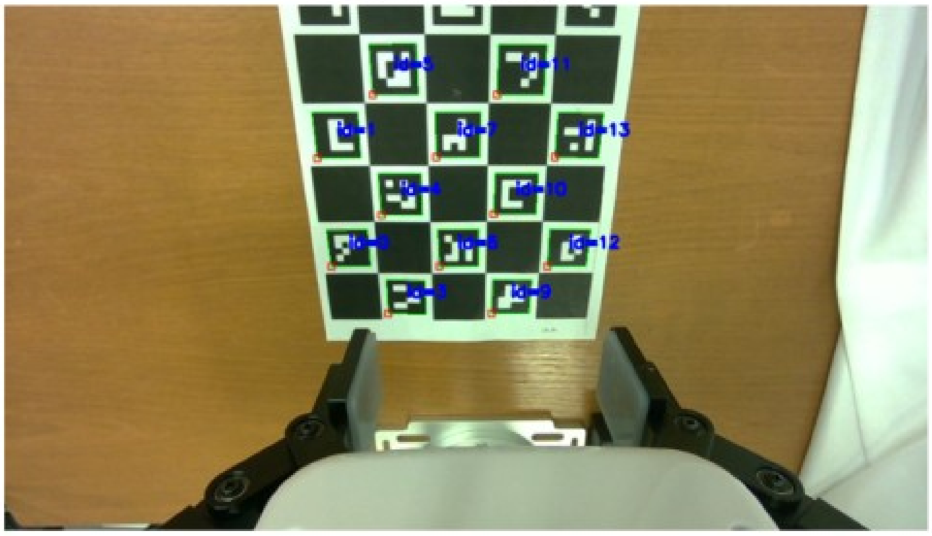
Detected markers



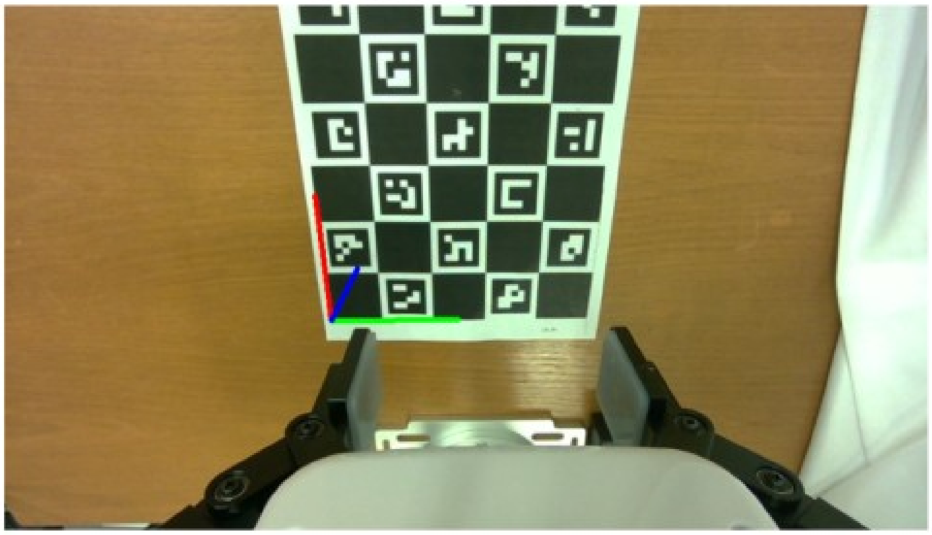
Pose Visualization



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)

```python
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)

```python
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
```

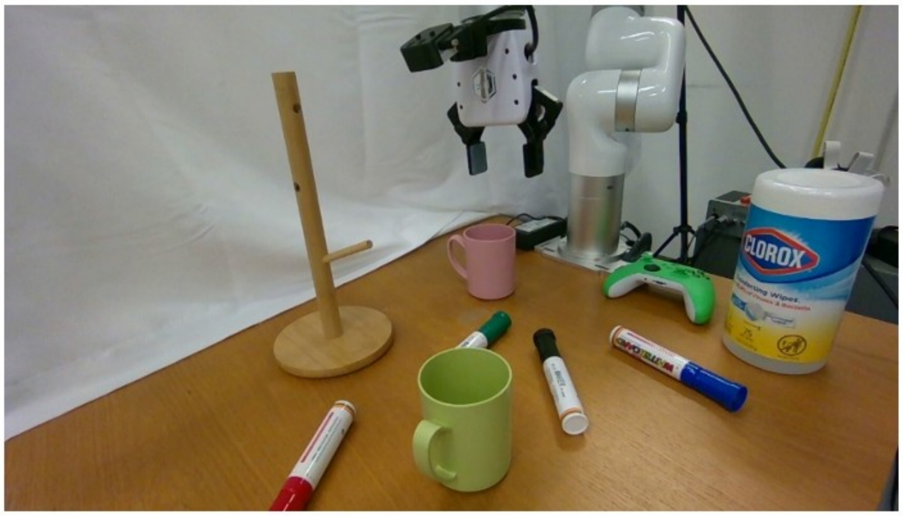
2.4 Visualize (10 pts)

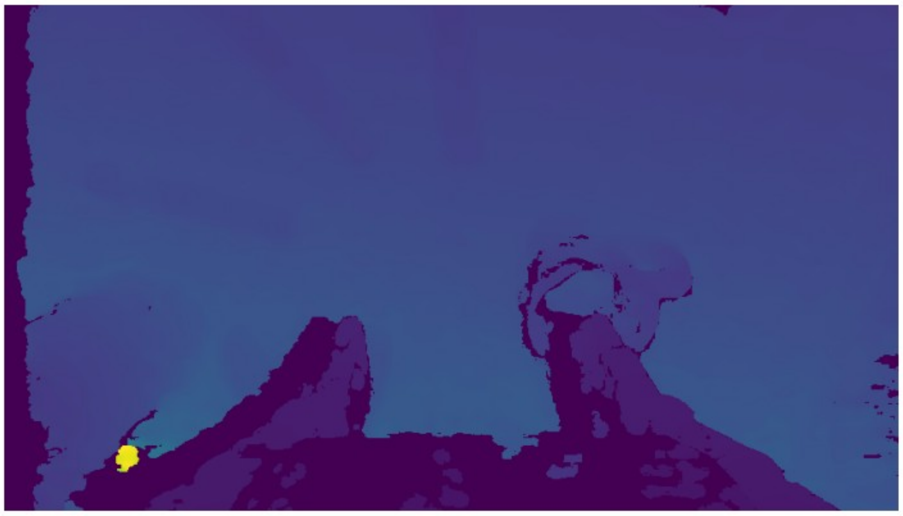
Finally, we are ready to visualize the point clouds! We will use the recording data from the `data/recordings/` dir. The code will show the two rgb and depth images at frame 0, and launch an open3d interactive visualization of the merged point cloud. Feel free to play around with other frames as well.

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])
```

```python
    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)

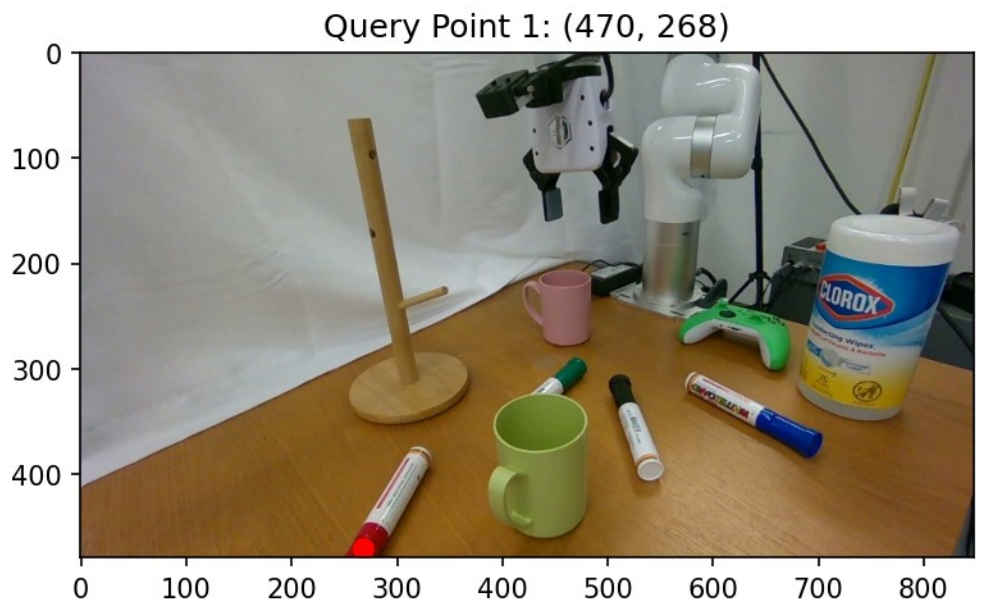
```
rgb_path = './data/recording/camera_front/rgb/000000.jpg'
rgb = cv2.imread(rgb_path)

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()
```

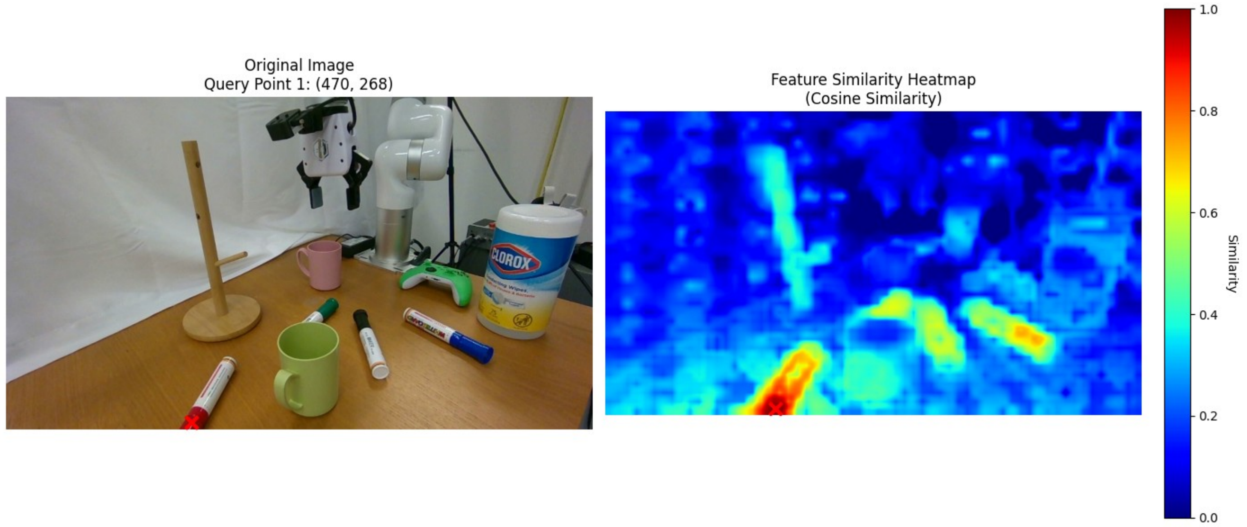
```
Using cache found in
/Users/jaiselsingh/.cache/torch/hub/facebookresearch_dinov2_main
```

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

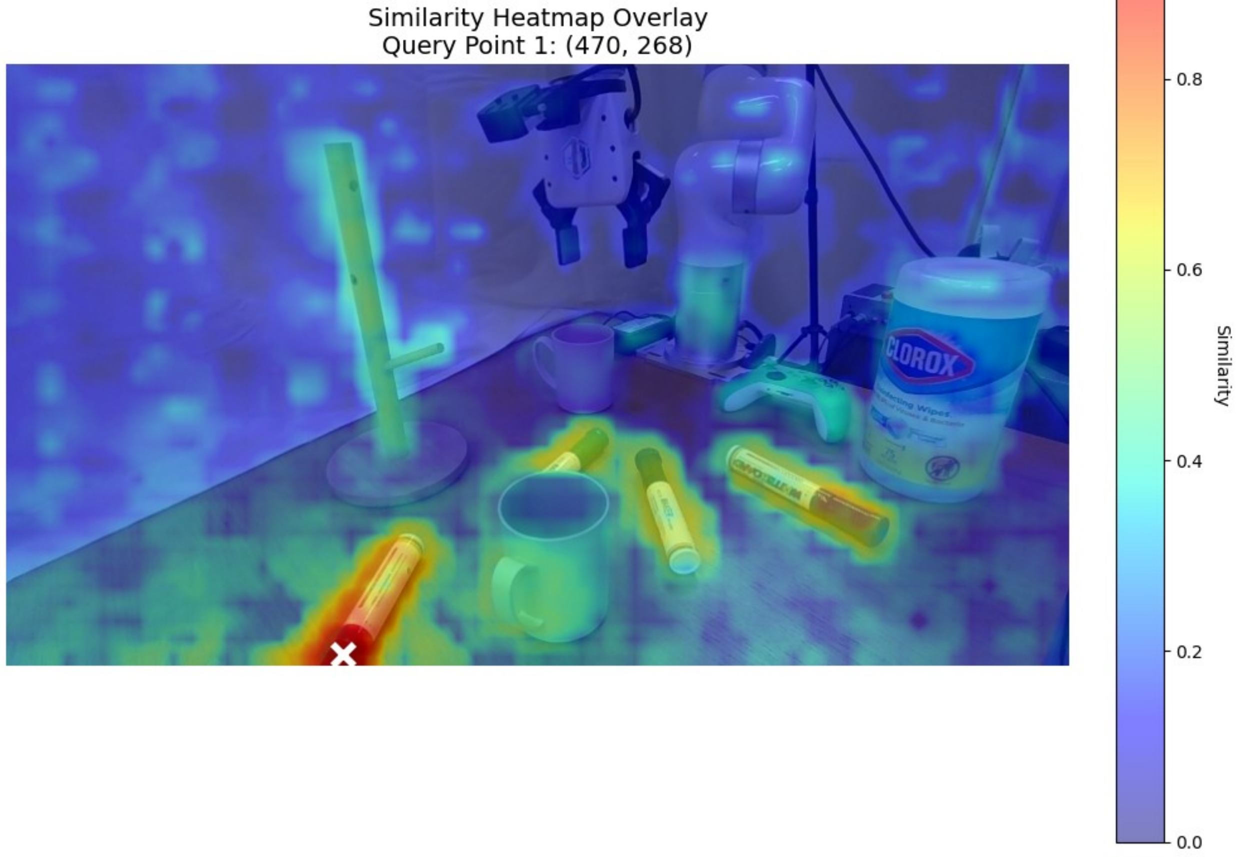


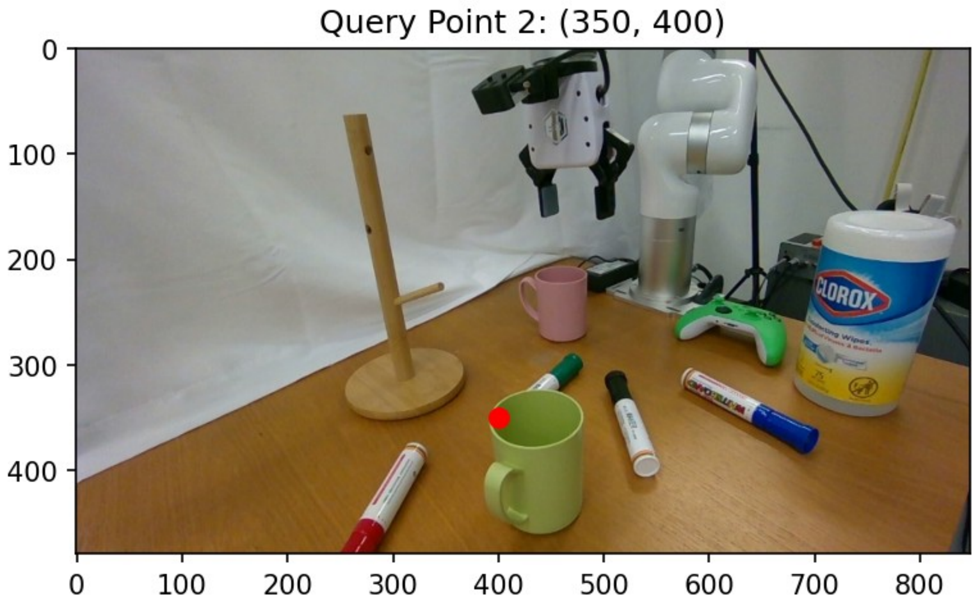
```
Similarity range: [-0.134, 1.000]
```

```
/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)
```

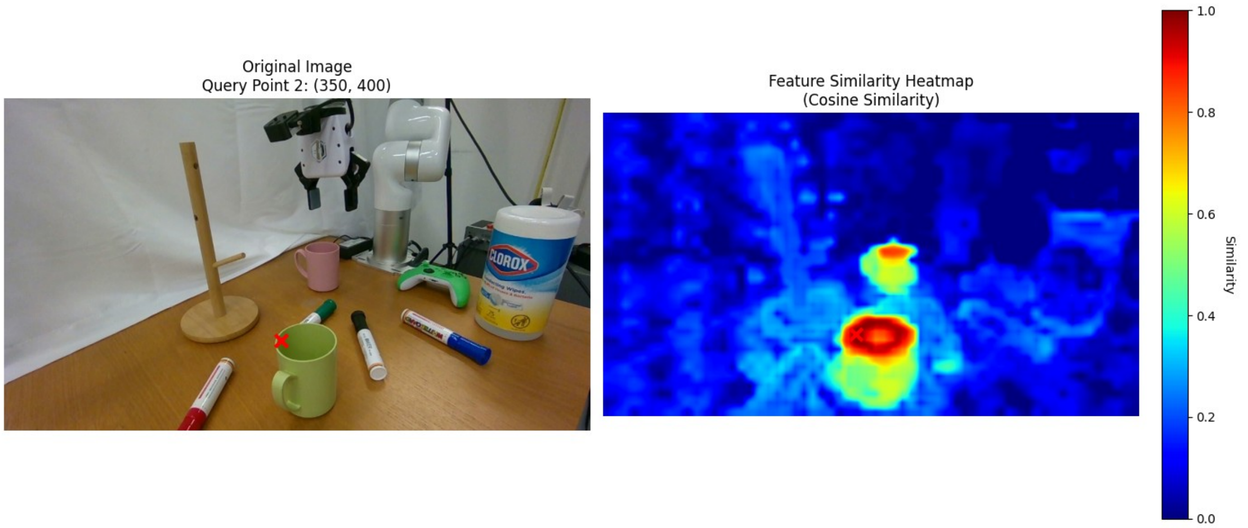


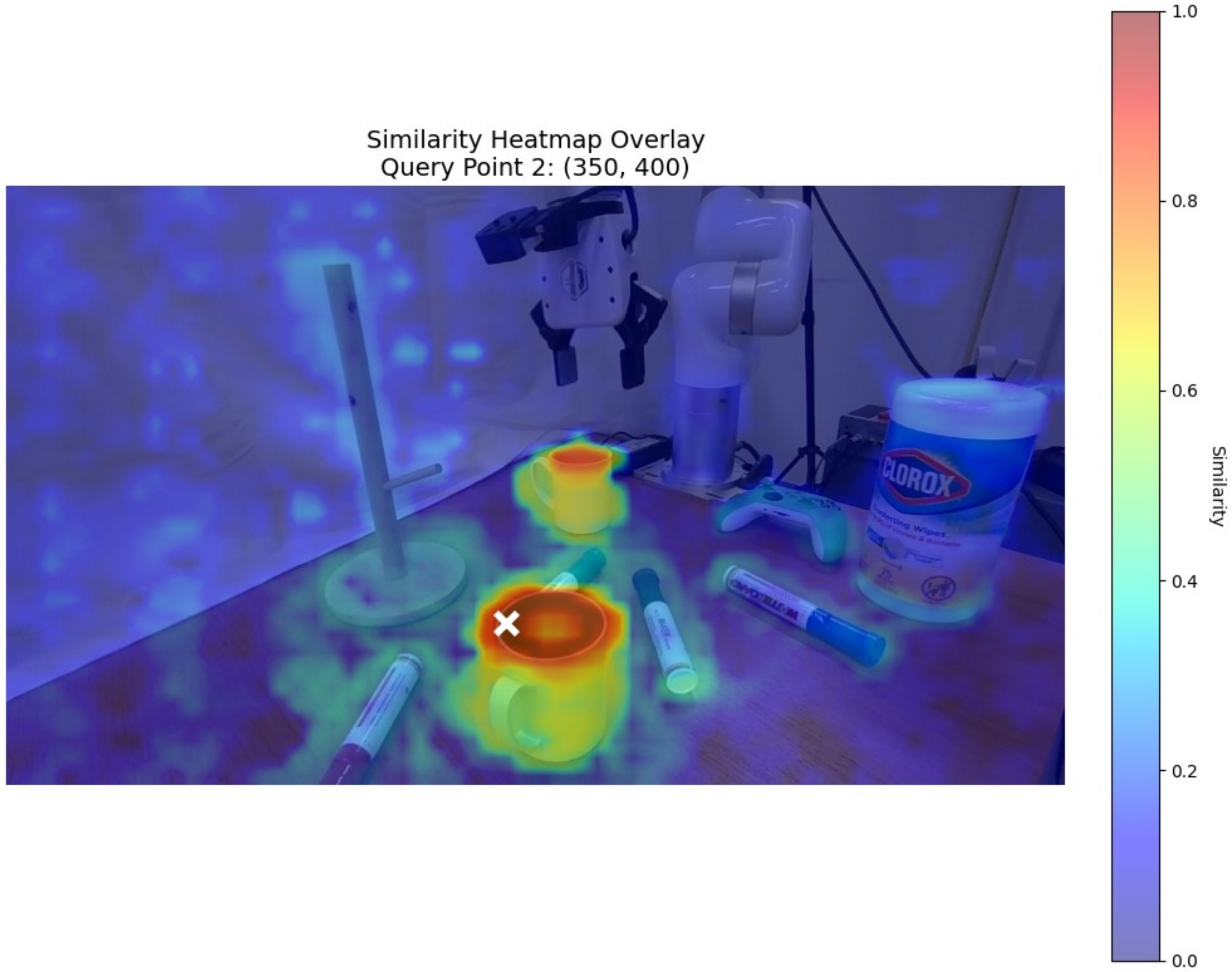
```
/var/folders/xk/f342x5yd6c77rjz1q6jc99jm0000gn/T/
ipykernel_17549/383732146.py:106: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').
Matplotlib is ignoring the edgecolor in favor of the facecolor.  This
behavior may change in the future.
  ax.scatter([y], [x],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

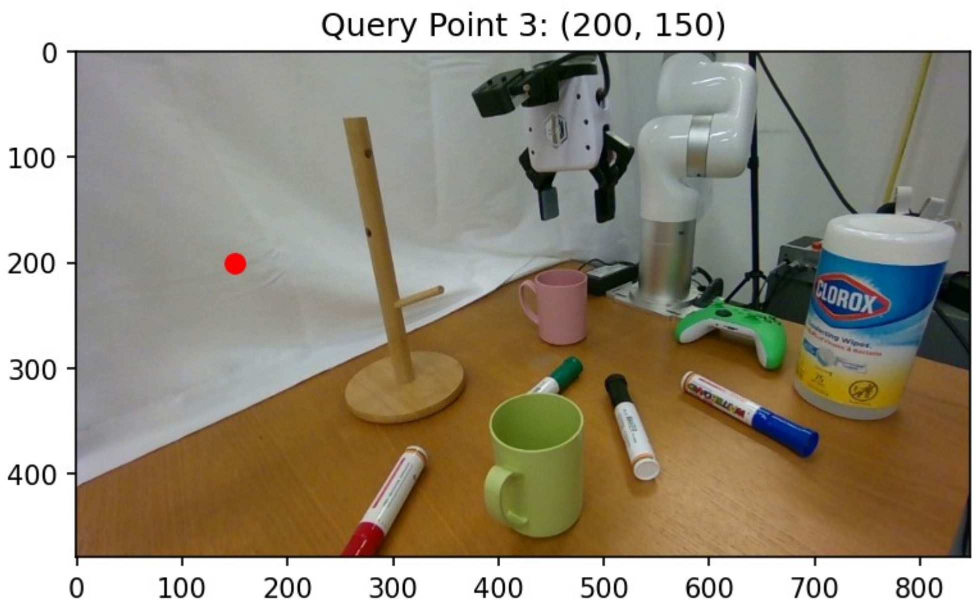

Similarity Heatmap Overlay
Query Point 1: (470, 268)
Similarity
1.0
0.8
0.6
0.4
0.2
0.0


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

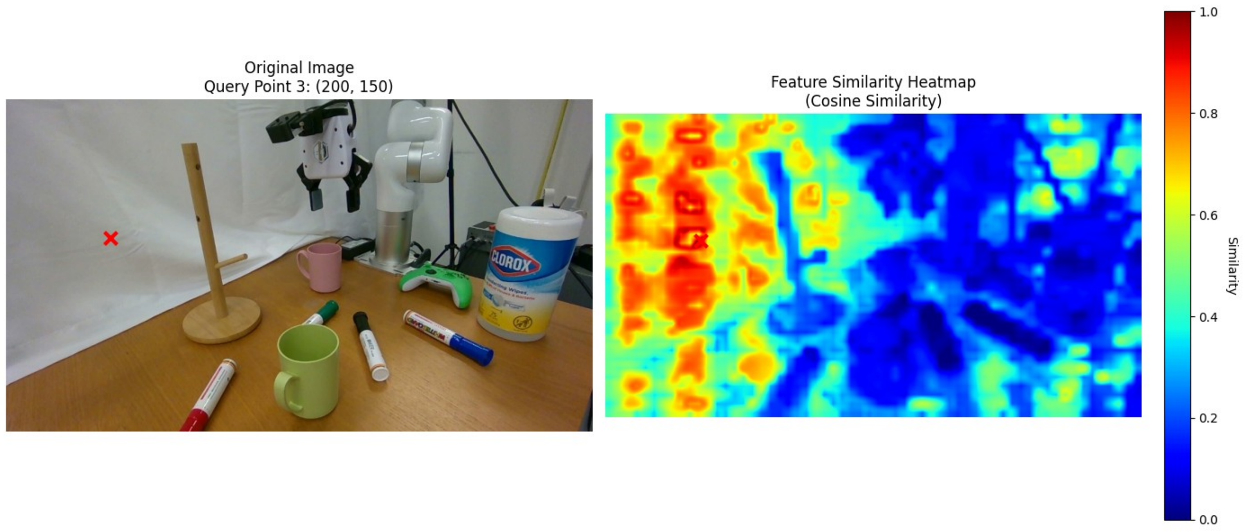
```
Similarity range: [-0.110, 1.000]
```

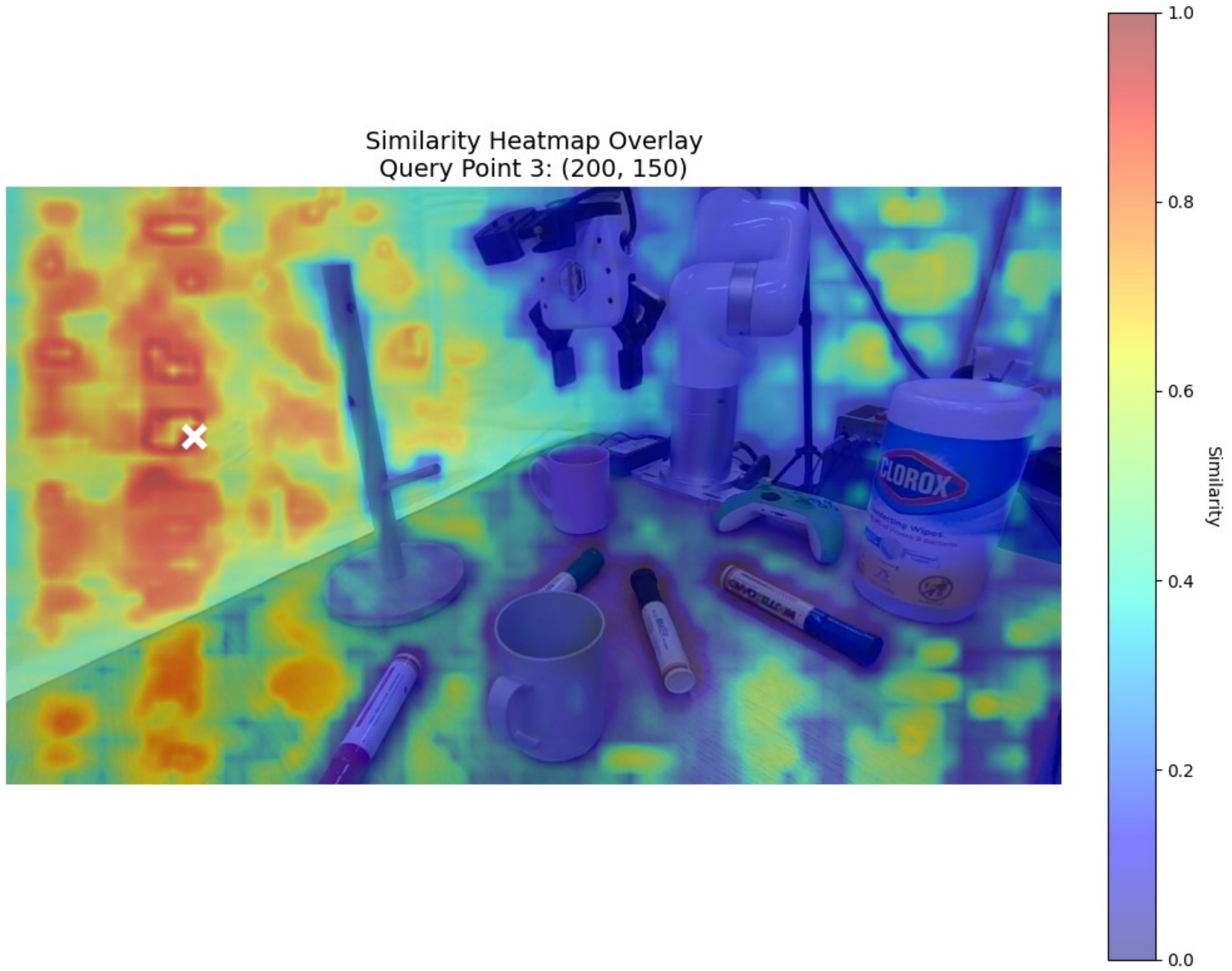






```
Similarity range: [-0.035, 1.000]
```





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`