Introduction

In this assignment you will practice calibrating a camera, computing homography between images and stitching panoramas. The goals of this assignment are as follows:

- Understand the intrinsic parameters of a prospective camera.
- Understand feature detection and feature matching process.
- Find the intrinsic and extrinsic parameters of a camera using ChArUco patterns.
- Compute homography between two images.
- Use homography to stitch panoramas.

Please fill in all the **TODO** blocks (including codes and texts). Most of the assignment can be done by calling functions in OpenCV. Once you are ready to submit:

- Export the notebook `CSCI677_spring25_assignment_1.ipynb` as a PDF `[Your USC ID]_CSCI677_spring25_assignment_1.pdf` and submit the PDF

Please make sure that the notebook have been run before exporting PDF, and your code and all cell outputs are visible in your submitted PDF. Regrading request will not be accepted if your code/output is not visible in the original submission. Thank you!

Calibration (35 pts)

Data Collection (5 points)

To determine the intrinsic parameters of a camera, you will need to capture sample images of an ChArUco board. These images are obtained by photographing printed ChArUco patterns displayed on a flat surface or screen.

Steps:

1. Use the provided sample code to generate your own ChArUco pattern. Feel free to adjust the function parameters.
2. Print the generated pattern or display it on a flat screen.
3. Capture at least **10 photos** of the ChArUco board from **different angles** to ensure accuracy in calibration.
4. Make sure that **each photo covers the entire ChArUco board** for proper detection.

These images will be used later for camera calibration.

In [1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

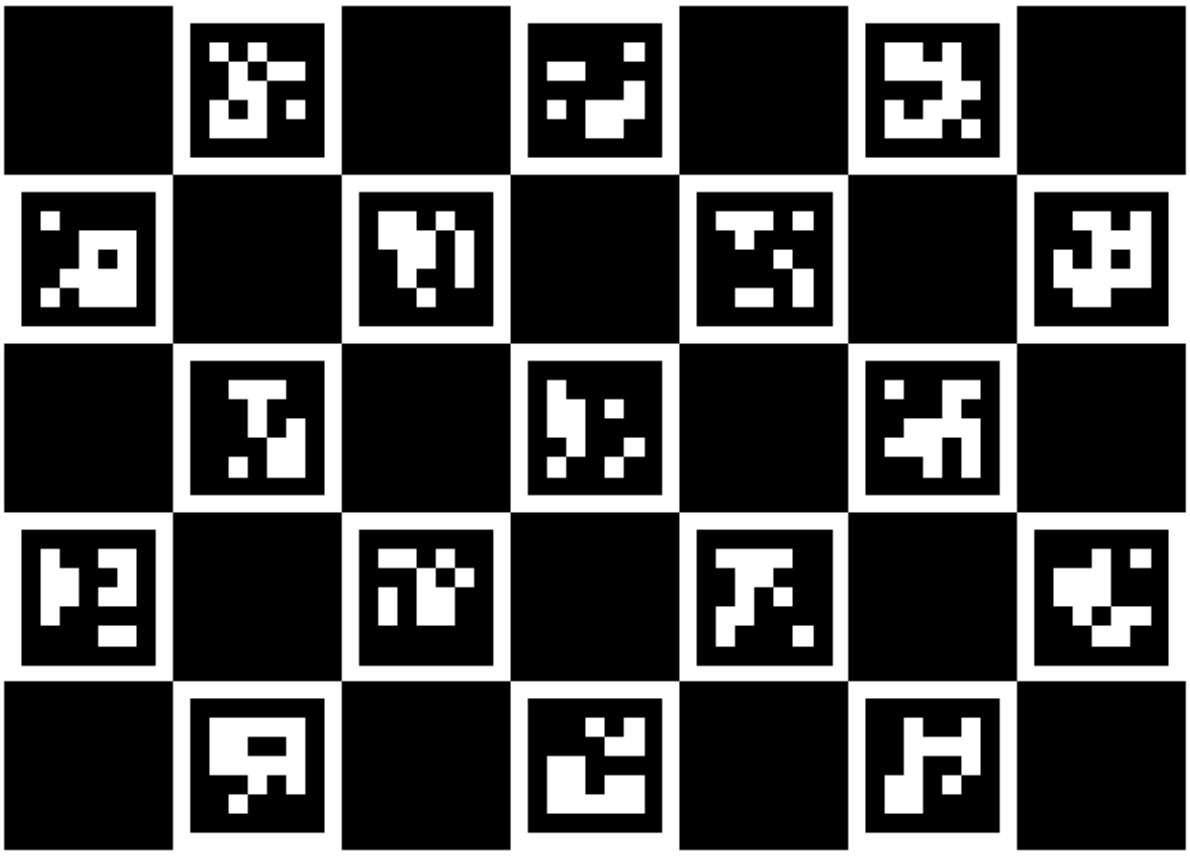
In [ ]:
```
!pip uninstall -y opencv-python opencv-contrib-python
!pip install opencv-contrib-python==4.6.0.66
```

```
WARNING: Skipping opencv-python as it is not installed.
Found existing installation: opencv-contrib-python 4.11.0.86
Uninstalling opencv-contrib-python-4.11.0.86:
  Successfully uninstalled opencv-contrib-python-4.11.0.86
Collecting opencv-contrib-python==4.6.0.66
  Downloading opencv_contrib_python-4.6.0.66-cp36-abi3-manylinux_2_17_x86_6
4.manylinux2014_x86_64.whl.metadata (18 kB)
Requirement already satisfied: numpy>=1.21.2 in /usr/local/lib/python3.11/di
st-packages (from opencv-contrib-python==4.6.0.66) (1.26.4)
Downloading opencv_contrib_python-4.6.0.66-cp36-abi3-manylinux_2_17_x86_64.m
anylinux2014_x86_64.whl (67.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 67.1/67.1 MB 8.2 MB/s eta 0:00:0
0
Installing collected packages: opencv-contrib-python
Successfully installed opencv-contrib-python-4.6.0.66
```

In [ ]:
```
import numpy as np
import cv2
from cv2 import aruco
import matplotlib.pyplot as plt
import os

def generate_charuco_board(
    save_dir: str,
    squares_x: int = 7,
    squares_y: int = 5,
    square_length: float = 1.0,
    marker_length: float = 0.8,
    aruco_dict_type: int = aruco.DICT_5X5_50,
    image_size: tuple = (2000, 2000),
    plot_results: bool = False
) -> tuple:
    """
    Generates and saves a ChArUco board image.

    Args:
        save_dir (str): Directory to save the generated ChArUco board image.
        squares_x (int): Number of squares along the horizontal direction.
        squares_y (int): Number of squares along the vertical direction.
        square_length (float): Side length of each square (in arbitrary unit
        marker_length (float): Side length of each marker (in arbitrary unit
        aruco_dict_type (int): Type of the ArUco dictionary to use.
        image_size (tuple): Size of the output image (width, height).
        plot_results (bool): Whether to display the generated board.

    Returns:
        tuple: The ArUco dictionary and ChArUco board object.
    """
```

```
    # Ensure the save directory exists
    os.makedirs(save_dir, exist_ok=True)

    # Create the ArUco dictionary
    aruco_dict = aruco.Dictionary_get(aruco_dict_type)

    # Create the ChArUco board
    board = aruco.CharucoBoard_create(
        squares_x, squares_y, square_length, marker_length, aruco_dict
    )

    # Generate the board image
    board_image = board.draw(image_size)

    # Save the image with a unique name
    filename = f"charuco_{aruco_dict_type}_{squares_x}x{squares_y}_{square_l
    filepath = os.path.join(save_dir, filename)
    cv2.imwrite(filepath, board_image)
    print(f"ChArUco board saved at: {filepath}")

    # Optionally plot the board
    if plot_results:
        plt.figure(figsize=(8, 8))
        plt.imshow(board_image, cmap='gray', interpolation="nearest")
        plt.axis("off")
        plt.show()

    return aruco_dict, board

# Example usage
save_directory = "./charuco_boards/"
aruco_dict, charuco_board = generate_charuco_board(
    save_dir=save_directory,
    plot_results=True
)
```

```
ChArUco board saved at: ./charuco_boards/charuco_4_7x5_1.0_0.8.tiff
```



In [ ]:
```
import cv2
print(cv2.__version__)
```

```
4.11.0
```

Calibration (15 pts)

To calibrate the camera, you will need to detect the ChArUco markers in the photos you captured during the data collection step. The following tutorial may be helpful.

Steps:

1. Detect markers: Use the `cv2.aruco.detectMarkers()` function to locate the ArUco markers in each image.
2. Calibrate the Camera: Use the `cv2.aruco.calibrateCameraCharuco()` function to calculate the intrinsic camera parameters. Provide the detected ChArUco

corners, corresponding IDs, board configuration, and image dimensions. You may find the following tutorial helpful.

3. Output the Intrinsic Parameters: Print the *__camera matrix__* and *__distortion coefficients__* after the calibration process.

In [ ]:
```python
# TODO
image_dir = "/content/drive/MyDrive/1acv/Arucoimgs"
images = [os.path.join(image_dir, filename) for filename in os.listdir(image

all_corners = []
all_ids = []
image_size = None
for img_path in images:
  img = cv2.imread(img_path)
  gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
  corners, ids, _ = cv2.aruco.detectMarkers(gray, aruco_dict)
  if ids is not None and len(ids)>0:
    retval, charuco_corners, charuco_ids = cv2.aruco.interpolateCornersCharu
    if retval > 0:
      all_corners.append(charuco_corners)
      all_ids.append(charuco_ids)
      if image_size is None:
        image_size = gray.shape[::-1]
  print(f"Detected markers in {len(all_corners)} images.")
  if len(all_corners) > 0:
    retval, camera_matrix, dist_coefs, rvecs, tvecs = cv2.aruco.calibrateCam
        all_corners, all_ids, charuco_board, image_size, None, None
    )
    print("\nResults: ")
    print("Camera Matrix:")
    print(camera_matrix)
    print("Distortion Coefficients:")
    print(dist_coefs)
```

```
Detected markers in 1 images.

Results:
Camera Matrix:
[[5.01219284e+03 0.00000000e+00 1.34201079e+03]
 [0.00000000e+00 4.41001947e+03 2.03910933e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Distortion Coefficients:
[[-1.79157120e+00  9.33375143e+01  5.43161099e-02 -3.01655039e-02
  -1.04196894e+03]]
Detected markers in 2 images.

Results:
Camera Matrix:
[[2.70174802e+03 0.00000000e+00 1.43186766e+03]
 [0.00000000e+00 2.74618777e+03 2.17707660e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Distortion Coefficients:
[[-1.64577950e-01  4.01811626e+00  3.23702935e-02 -8.41660972e-03
  -1.36208567e+01]]
Detected markers in 3 images.

Results:
Camera Matrix:
[[2.58456756e+03 0.00000000e+00 1.40816603e+03]
 [0.00000000e+00 2.62120688e+03 2.19593736e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Distortion Coefficients:
[[-1.41121856e-01  2.89911748e+00  2.53849651e-02  2.70793176e-04
  -8.84978121e+00]]
Detected markers in 4 images.

Results:
Camera Matrix:
[[2.65380561e+03 0.00000000e+00 1.41703600e+03]
 [0.00000000e+00 2.66865311e+03 2.08012490e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Distortion Coefficients:
[[ 0.10807855  0.2405673   0.00550598 -0.00310303 -0.99659577]]
Detected markers in 5 images.

Results:
Camera Matrix:
[[3.05545816e+03 0.00000000e+00 1.58919967e+03]
 [0.00000000e+00 3.04822392e+03 1.95195927e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Distortion Coefficients:
[[ 5.31674575e-01 -6.51042734e+00 -1.35383036e-02  8.85458708e-03
   2.69675515e+01]]
Detected markers in 6 images.

Results:
Camera Matrix:
[[2.97852784e+03 0.00000000e+00 1.49889361e+03]
 [0.00000000e+00 2.97350690e+03 2.00250078e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
```

```
Distortion Coefficients:
[[ 3.05886033e-01 -3.02139457e+00 -6.58433965e-03 -3.51268361e-04
   1.19605966e+01]]
Detected markers in 7 images.

Results:
Camera Matrix:
[[2.89889344e+03 0.00000000e+00 1.46727402e+03]
 [0.00000000e+00 2.91117511e+03 1.97194373e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Distortion Coefficients:
[[ 6.40388802e-01 -6.18716072e+00 -1.39016608e-02 -1.41007686e-02
   1.52744848e+01]]
Detected markers in 8 images.

Results:
Camera Matrix:
[[2.95489589e+03 0.00000000e+00 1.47109216e+03]
 [0.00000000e+00 2.95806305e+03 1.98109778e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Distortion Coefficients:
[[ 5.24478266e-01 -5.30916783e+00 -1.22913249e-02 -9.17781635e-03
   1.26207250e+01]]
Detected markers in 9 images.

Results:
Camera Matrix:
[[2.95681671e+03 0.00000000e+00 1.45998128e+03]
 [0.00000000e+00 2.96055543e+03 1.98038130e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Distortion Coefficients:
[[ 0.48797552 -4.74081347 -0.01221758 -0.01125148 10.57543711]]
Detected markers in 10 images.

Results:
Camera Matrix:
[[3.02010556e+03 0.00000000e+00 1.49508567e+03]
 [0.00000000e+00 3.03183073e+03 1.99889798e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Distortion Coefficients:
[[ 0.35021089 -3.31734896 -0.00806067 -0.01086652  5.64958305]]
Detected markers in 11 images.

Results:
Camera Matrix:
[[3.04204653e+03 0.00000000e+00 1.50444879e+03]
 [0.00000000e+00 3.04973836e+03 1.99817002e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Distortion Coefficients:
[[ 0.35751351 -3.49159298 -0.0071574  -0.00864811  5.94197299]]
Detected markers in 12 images.

Results:
Camera Matrix:
[[3.03677736e+03 0.00000000e+00 1.49936000e+03]
 [0.00000000e+00 3.04230457e+03 1.99905056e+03]
```

```
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Distortion Coefficients:
[[ 0.37726715 -3.71279008 -0.00693506 -0.00889132  6.79081887]]
```

Pose Estimation (10 pts)

Once you have obtained the camera parameters, use them to calculate the pose of the Charuco board. To validate your code, render the axes on 3 of your Charuco board images. You can refer to the following tutorial for guidance on pose estimation and rendering the axes.

In [ ]:
```python
# TODO
selected_images = images[:12]  # You can change the indices if needed

# Define axis length for rendering
axis_length = 3  # Adjust based on board size

for img_path in selected_images:
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    corners, ids, _ = cv2.aruco.detectMarkers(gray, aruco_dict)
    if ids is not None and len(ids) > 0:
        retval, charuco_corners, charuco_ids = cv2.aruco.interpolateCornersC
        if retval > 0:
            # Estimate camera pose
            retval, rvec, tvec = cv2.aruco.estimatePoseCharucoBoard(charuco_
            if retval > 0:
                success, rvec, tvec = cv2.aruco.estimatePoseCharucoBoard(cha
                if success:
                    # Draw axes on the image
                    cv2.drawFrameAxes(img, camera_matrix, dist_coefs, rvec,

                    plt.figure(figsize=(8, 8))
                    plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
                    plt.title(f"Image: {img_path}")
                    plt.axis("off")
                    plt.show()
```

Image: /content/drive/MyDrive/1acv/Arucoimgs/1.JPG

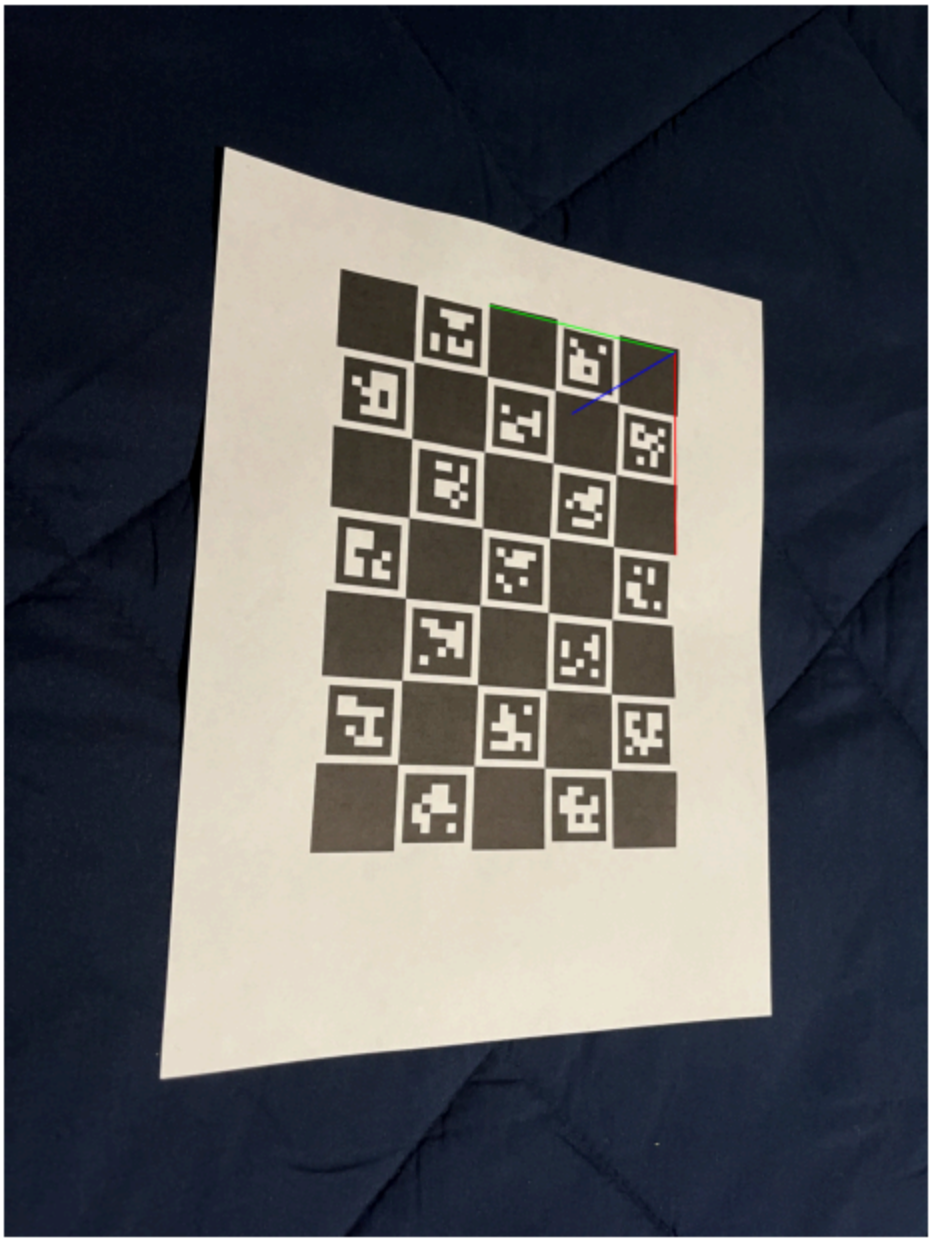


Image: /content/drive/MyDrive/1acv/Arucoimgs/2.JPG

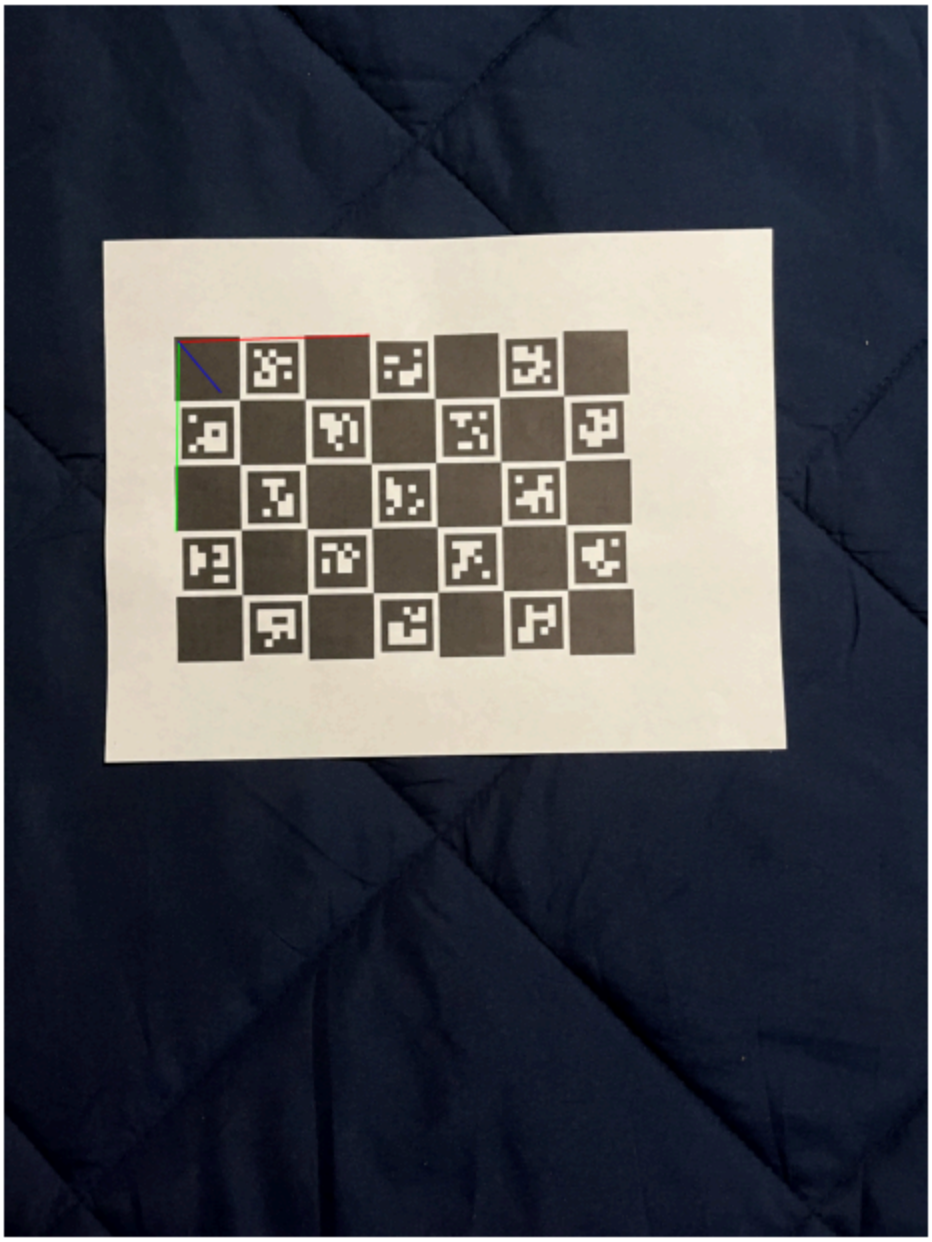


Image: /content/drive/MyDrive/1acv/Arucoimgs/3.JPG

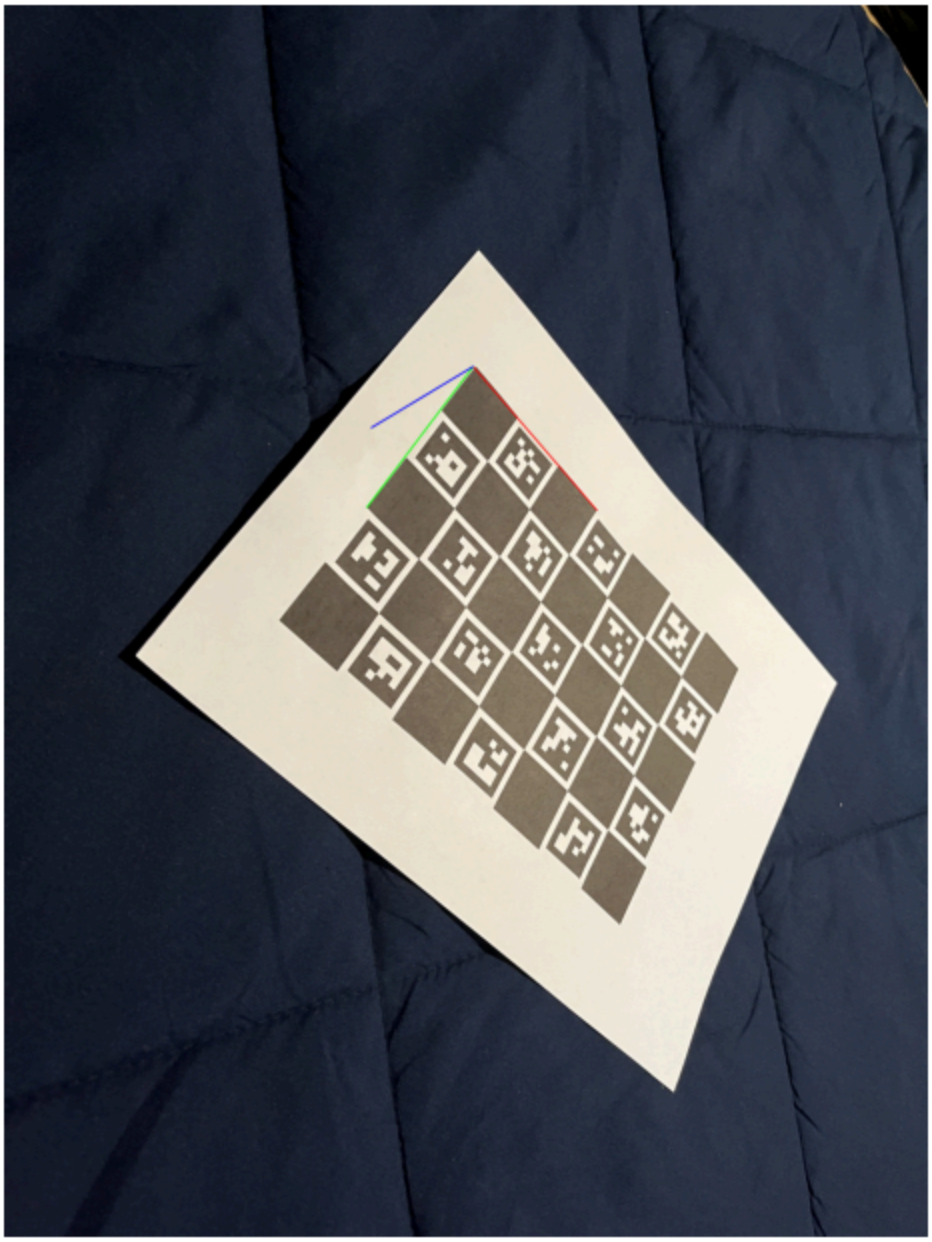


Image: /content/drive/MyDrive/1acv/Arucoimgs/4.JPG

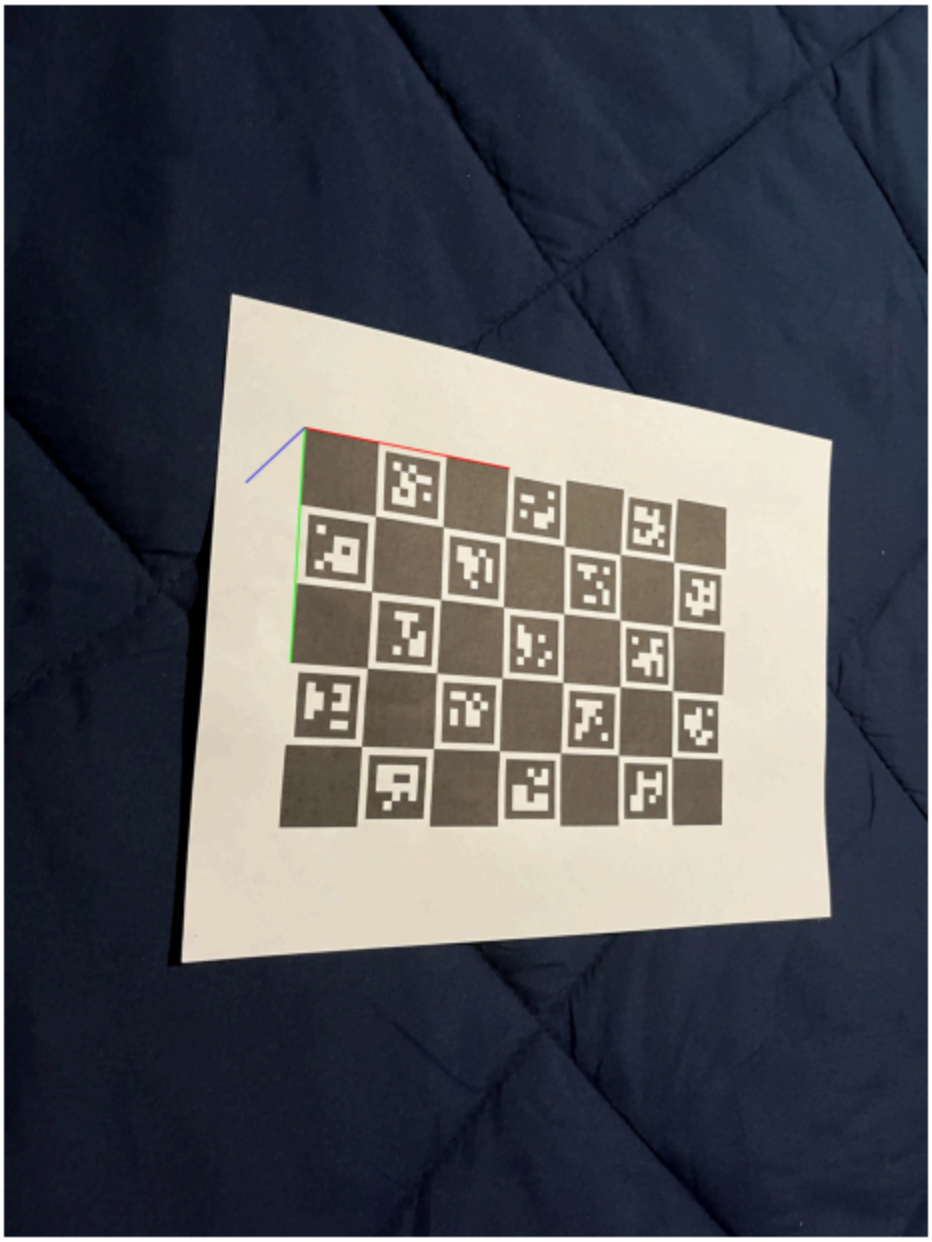


Image: /content/drive/MyDrive/1acv/Arucoimgs/5.JPG

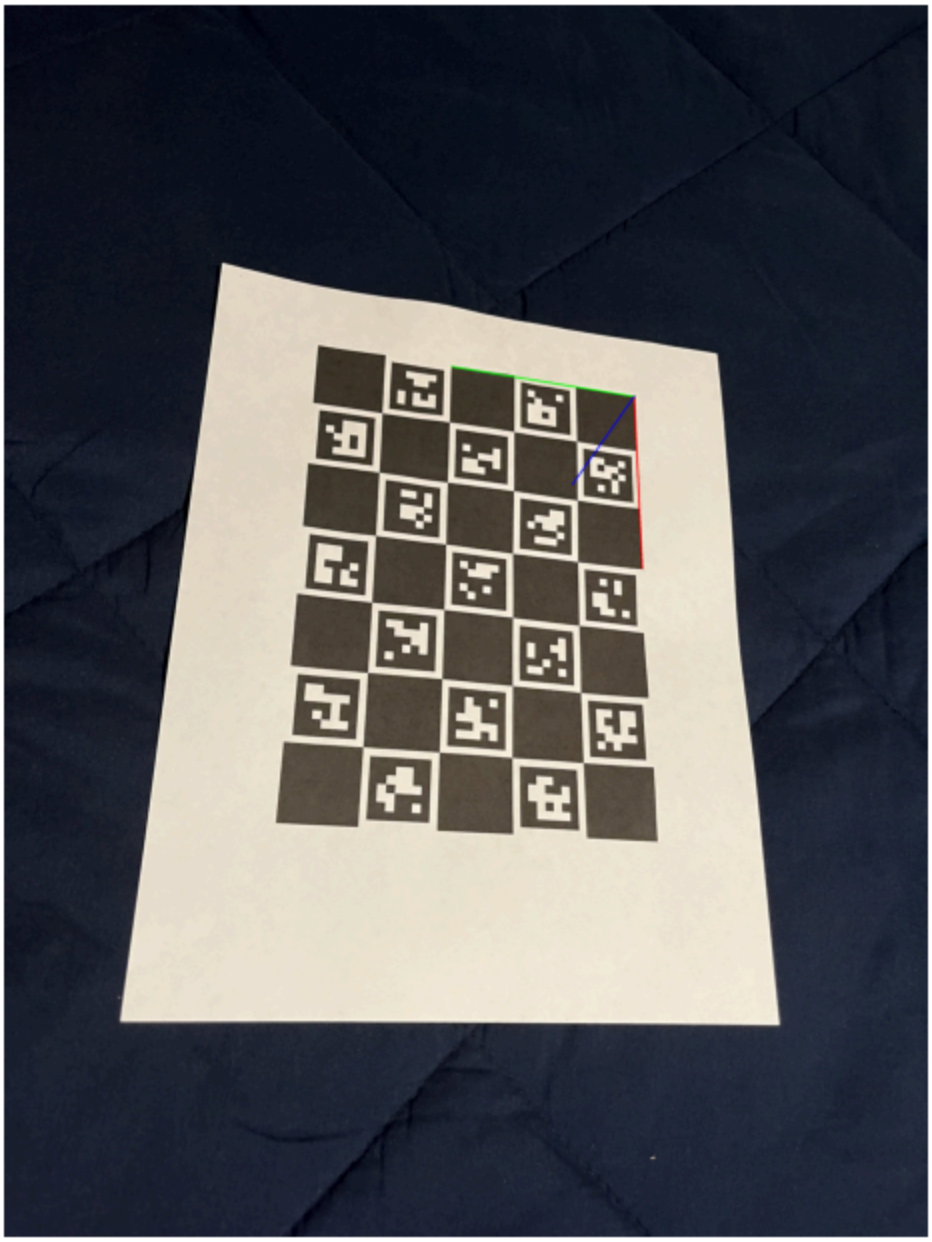


Image: /content/drive/MyDrive/1acv/Arucoimgs/6.JPG

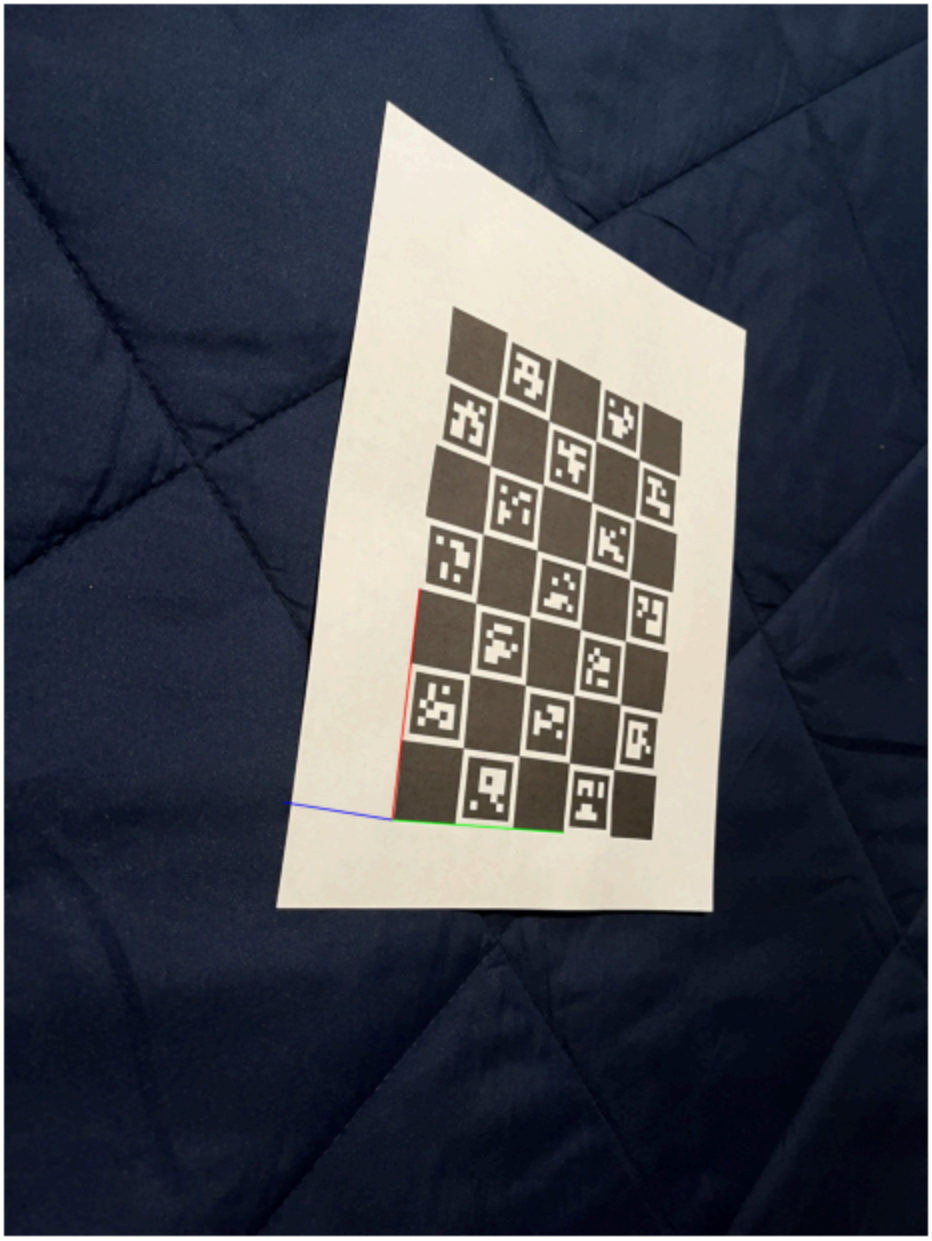


Image: /content/drive/MyDrive/1acv/Arucoimgs/7.JPG

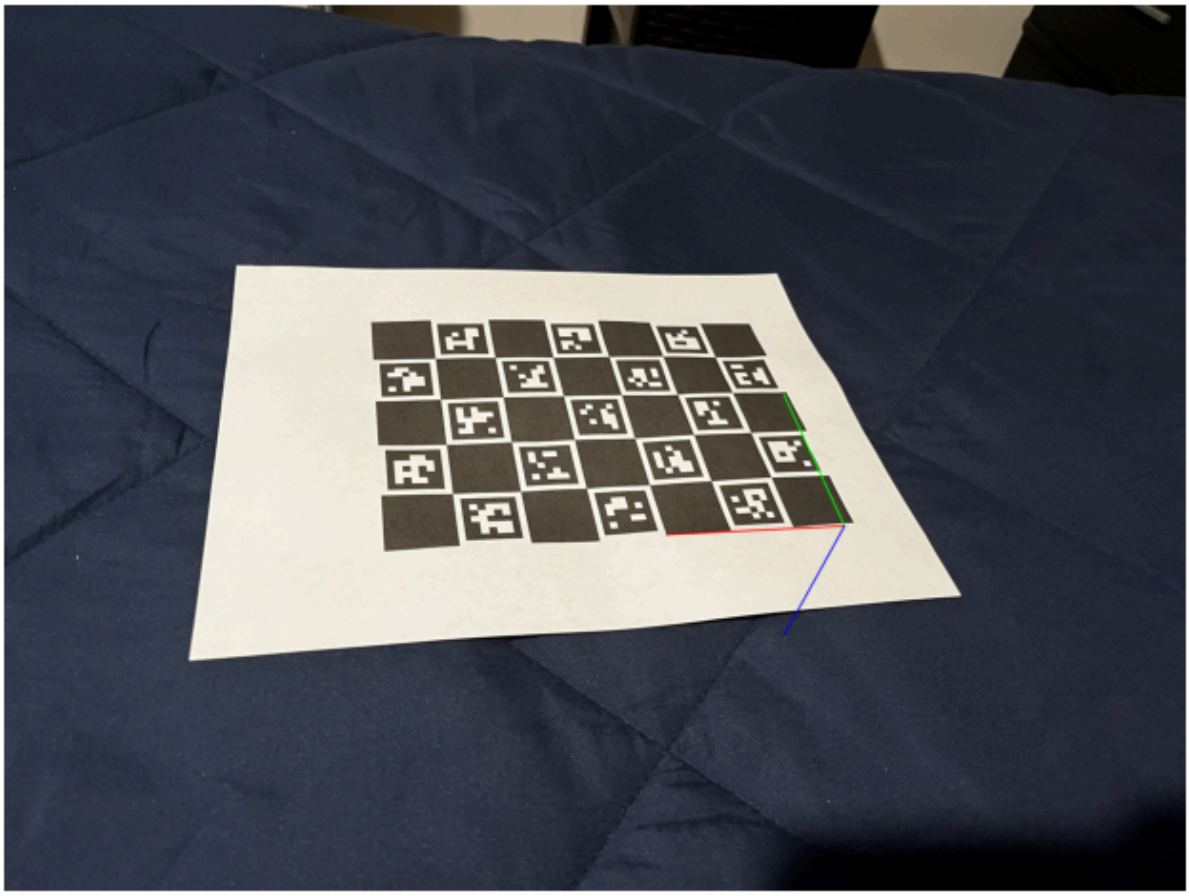


Image: /content/drive/MyDrive/1acv/Arucoimgs/8.JPG

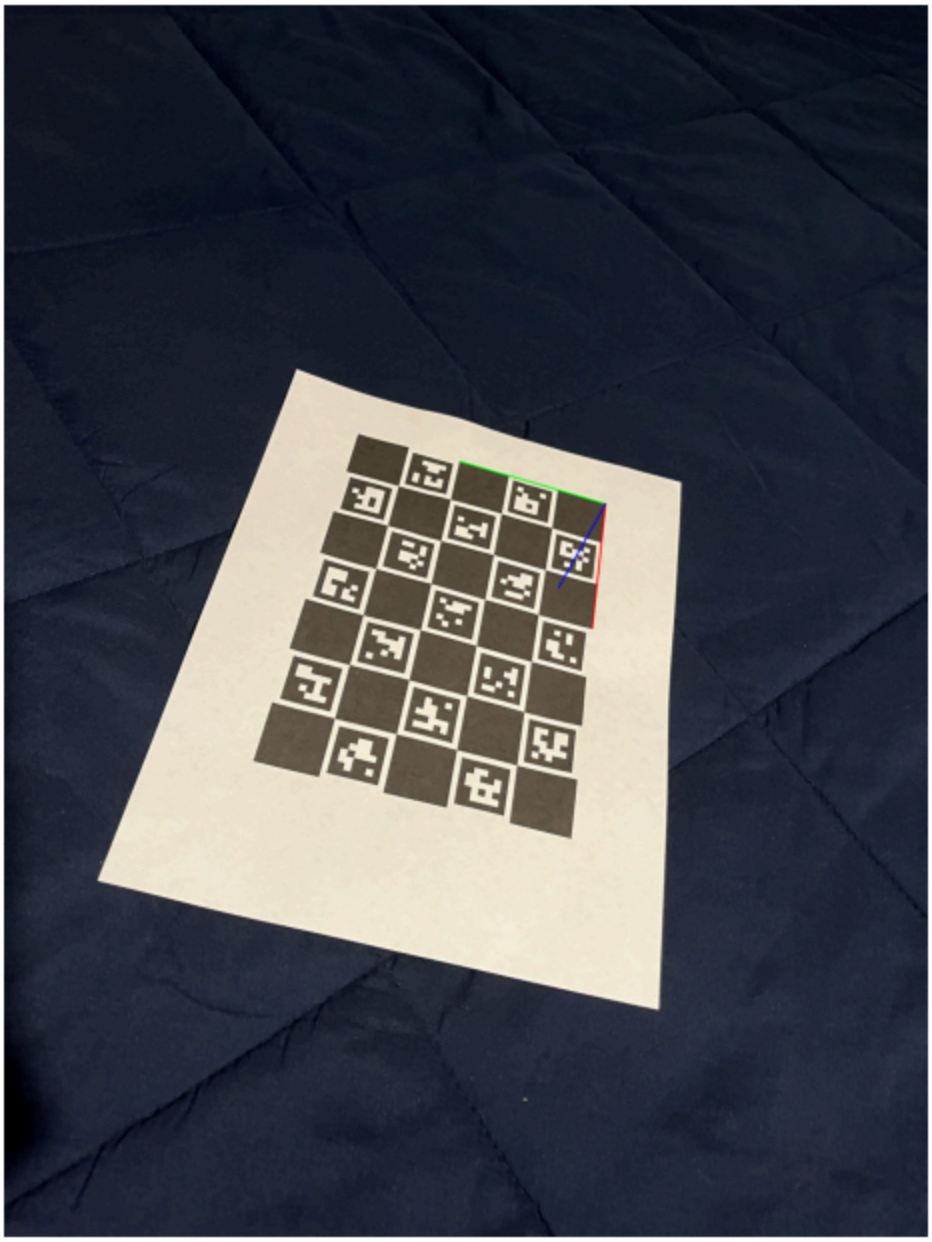


Image: /content/drive/MyDrive/1acv/Arucoimgs/9.JPG

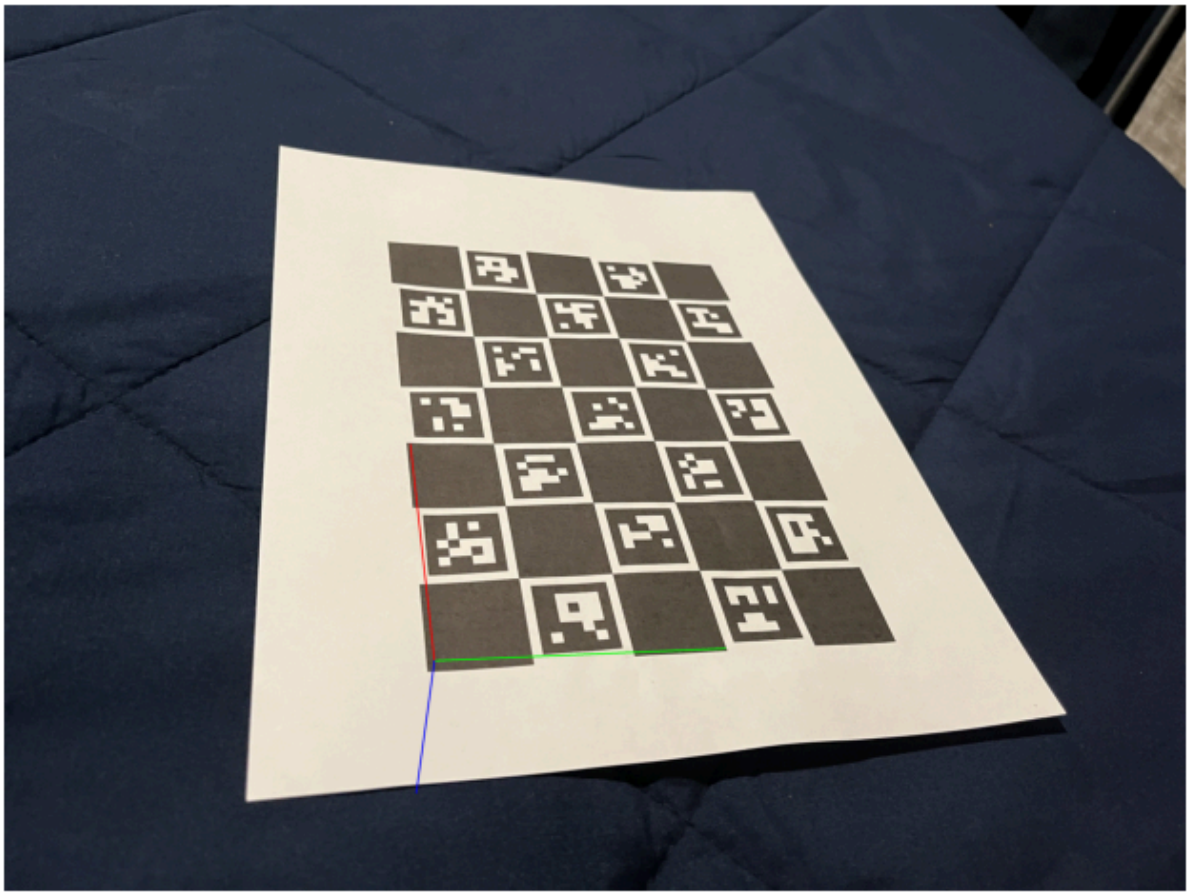


Image: /content/drive/MyDrive/1acv/Arucoimgs/10.JPG

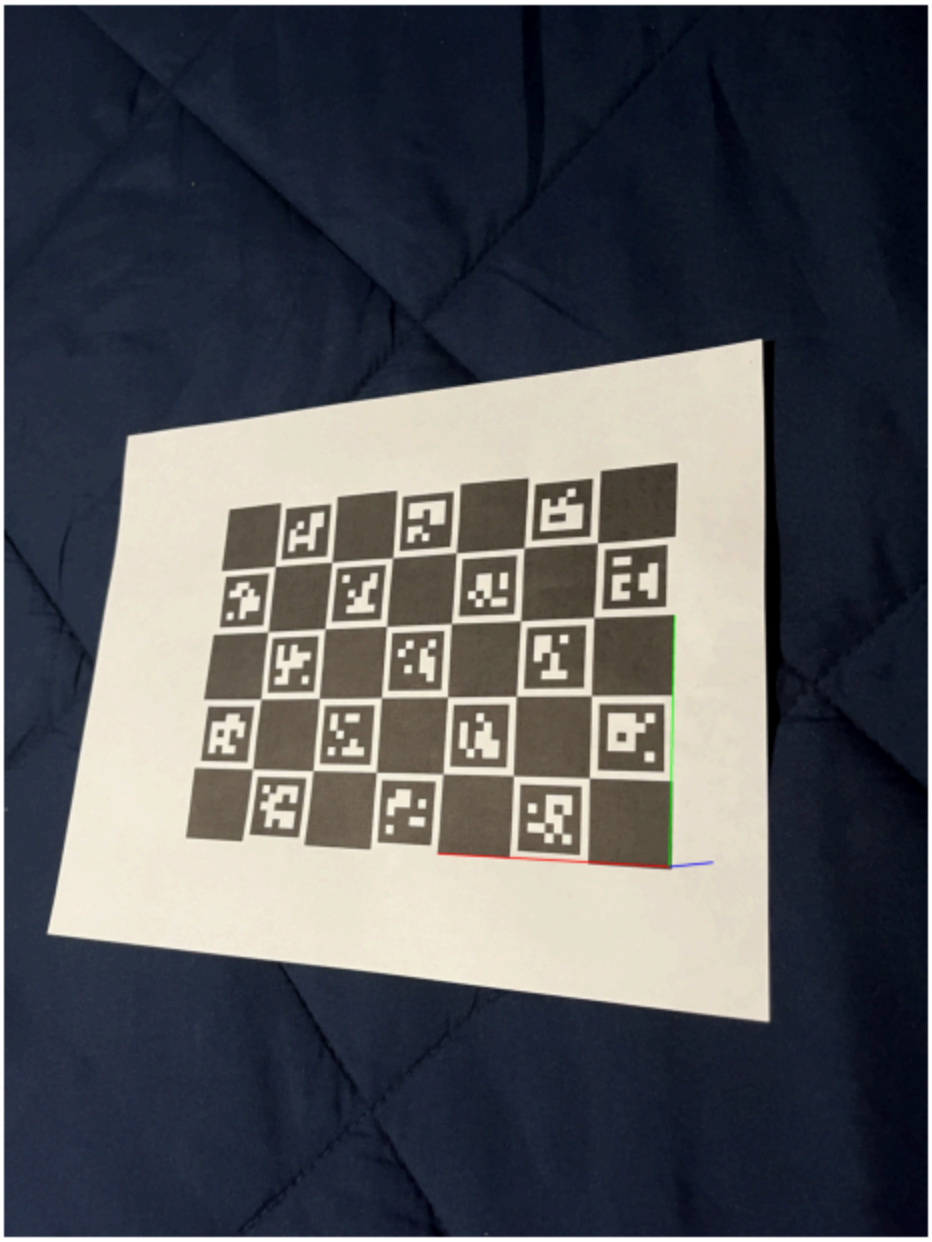


Image: /content/drive/MyDrive/1acv/Arucoimgs/11.JPG

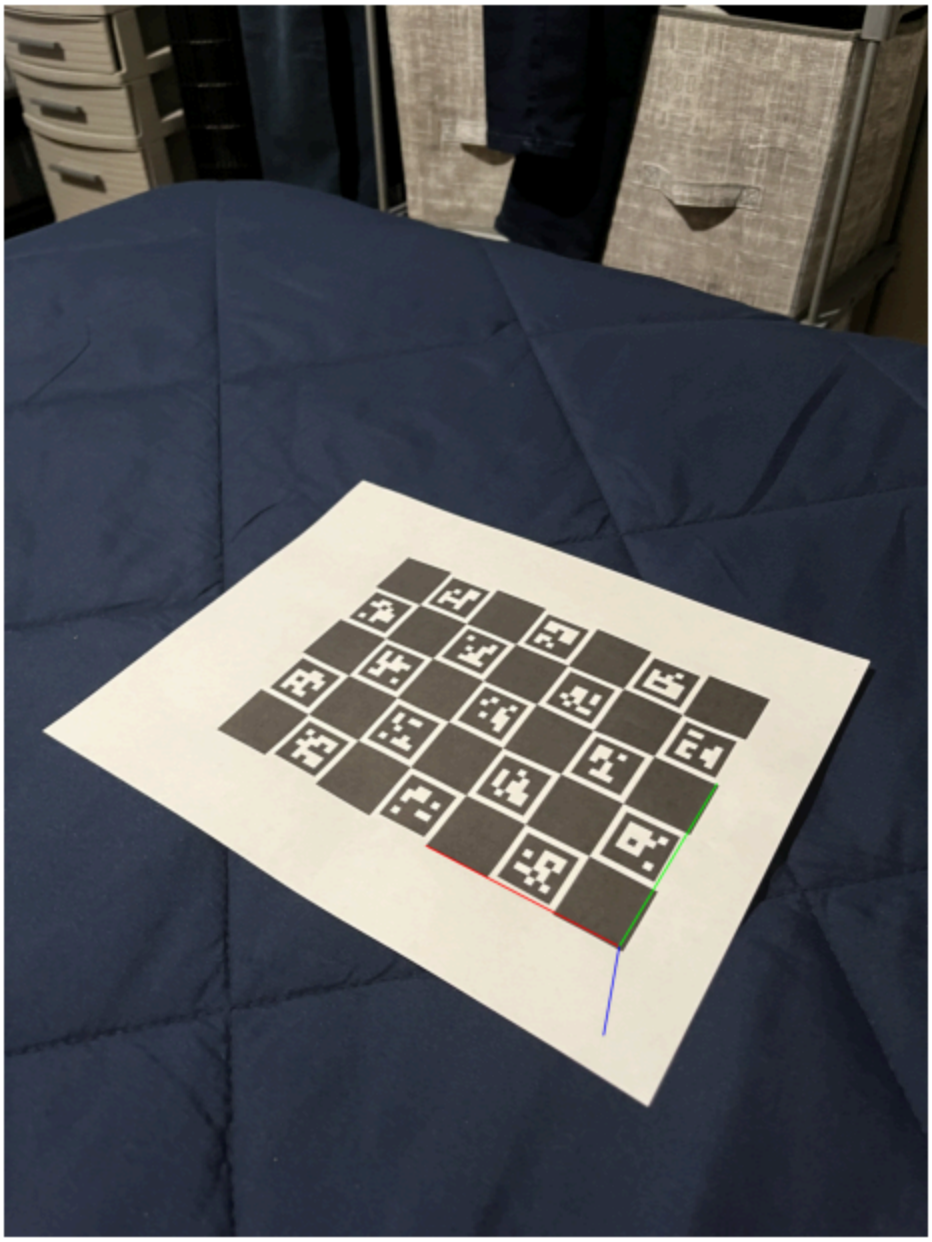
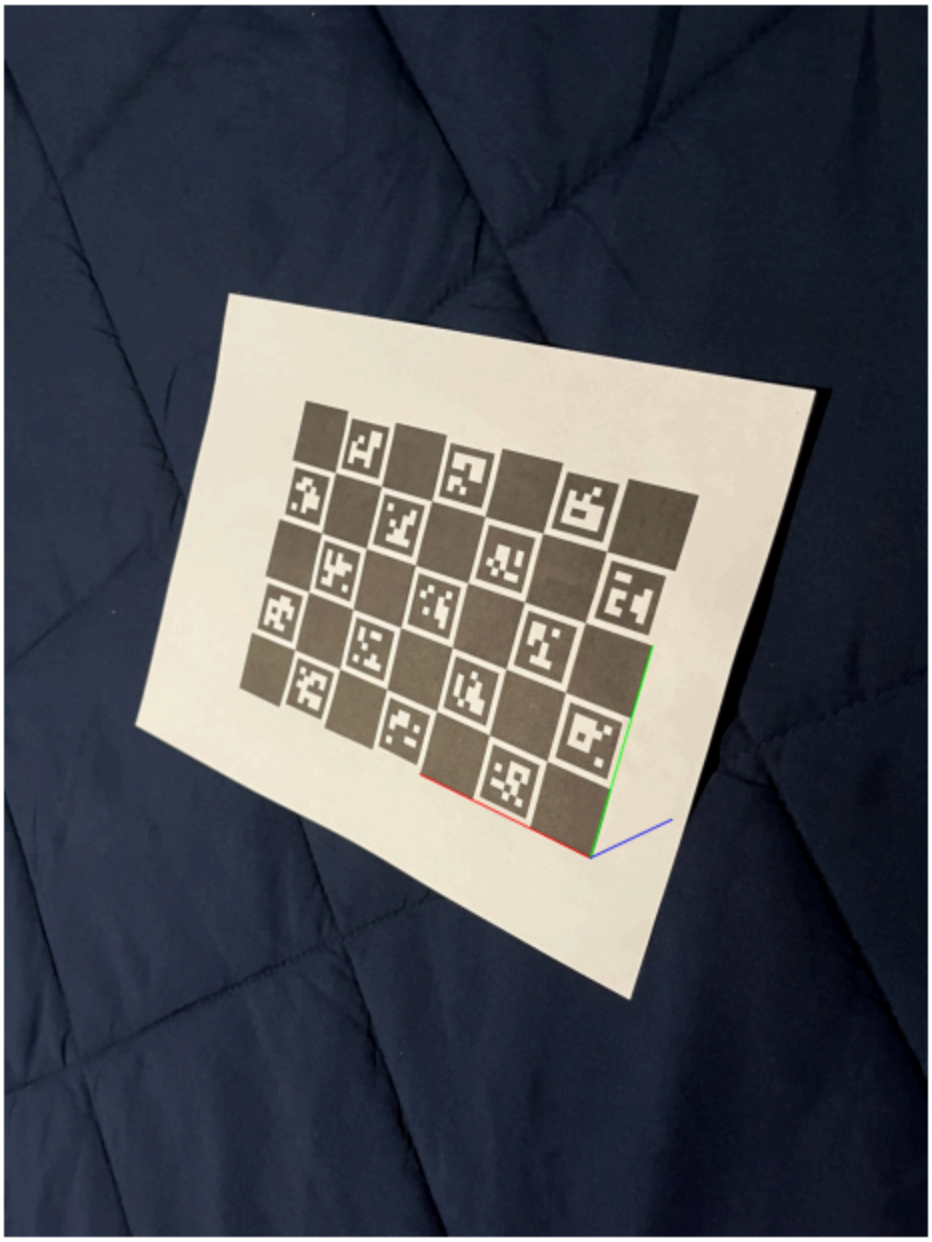


Image: /content/drive/MyDrive/1acv/Arucoimgs/12.JPG



Observation (5 pts)

Write down your observations regarding the results you obtained throughout the `Calibration` section.

Observations are as follows:

1. Number of Images Used: The calibration process used a 12 images in my case. The more images with good marker detection the

better calibration results. Ideally, more than 10 images should be used.

2. Camera Matrix and Distortion Coefficients: The camera matrix and distortion coefficients were successfully calculated and printed for each set of images where markers were detected. The quality of these parameters directly affects the accuracy of the subsequent pose estimation step.

3. Marker Detection Quality: Images with more markers contribute more reliably to the calibration. Issues like blur, low lighting or the ChArUco board being partially out of frame can lead to poor detection.

4. Calibration Accuracy: The quality of the calibration is determined by the accuracy of the detected markers. A more robust calibration process may be needed if images contain less markers, or if the marker detection is noisy.

5. In some pictures when there is slack in the paper, calibration processes are not able to adjust with it.

Homography (35 pts)

Data Preparation

Please download the `data.zip` file from the link, and extract it to a location of your choice. For this part of the assignment, use the images in the `homography` folder.

Feature Detection (10 pts)

After you have the two photos, load them using `cv.imread()`. Convert them to grayscale images. Create a SURF feature detector and detect keypoints on all images. Display the keypoints with size and orientation. For image display (only), adjust the Hessian threshold to show fewer than 30 features.

You can follow the tutorial here to understand the implementation and use of SURF.

In [ ]:
```
# prompt: uninstall opencv and contrib

!pip uninstall opencv-python
!pip uninstall opencv-contrib-python
```

In [ ]:
```
# !sudo apt-get update
# !sudo apt-get install -y build-essential cmake git libgtk2.0-dev pkg-confi

!git clone https://github.com/opencv/opencv.git
!git clone https://github.com/opencv/opencv_contrib.git
```

In [ ]:
```
%cd opencv
%mkdir build
%cd build
!cmake -DOPENCV_EXTRA_MODULES_PATH=../../opencv_contrib/modules \
      -DOPENCV_ENABLE_NONFREE=ON ..
!make -j$(nproc)
!sudo make install
```

In [4]:
```
import cv2
print(cv2.__version__)
```

```
4.12.0-dev
```

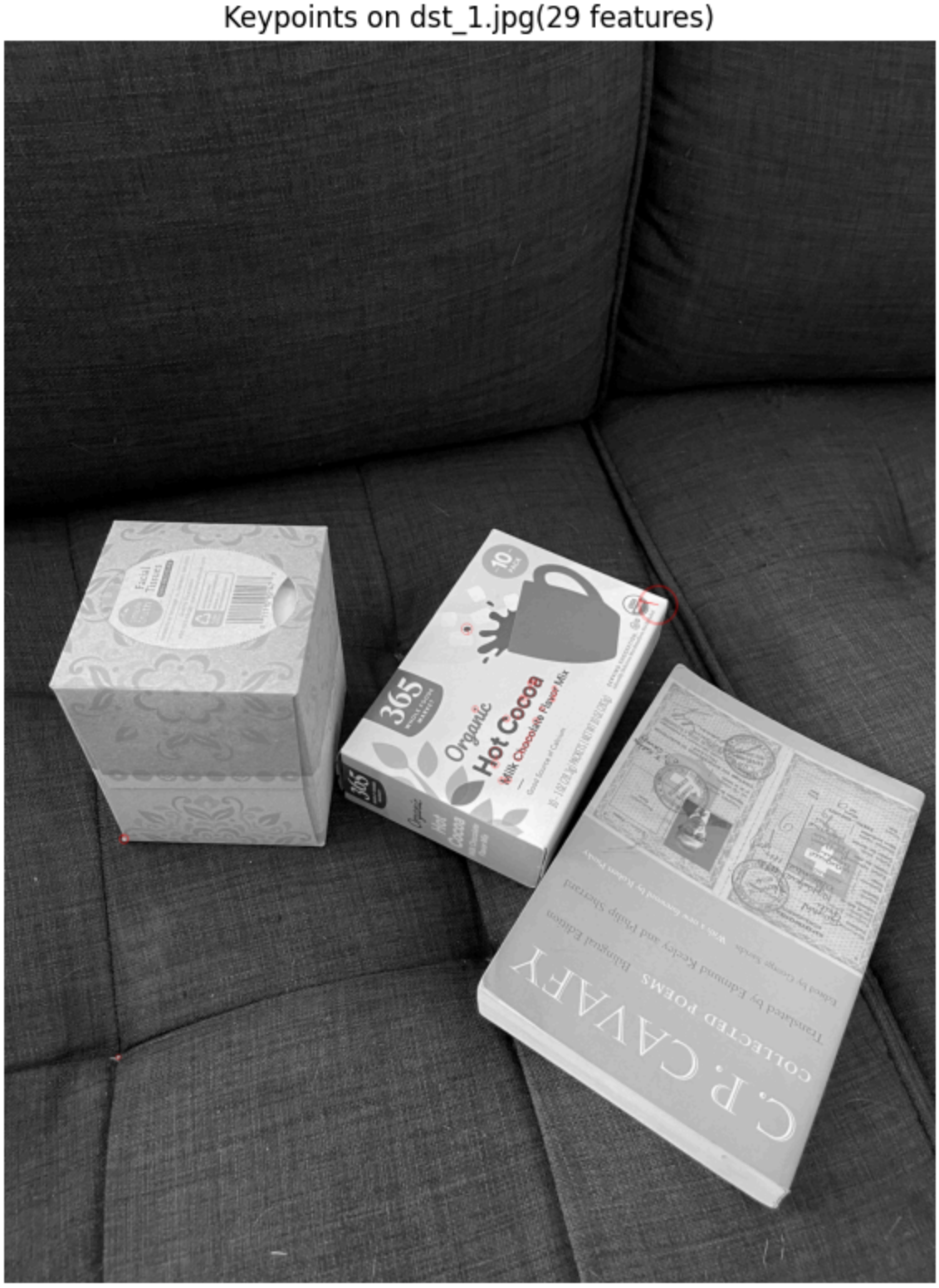
In [25]:
```
# TODO
import os
homography_loc = "/content/drive/MyDrive/1acv/csci677_2025_assignment1_data/
import cv2
import os
from matplotlib import pyplot as plt
img1 = cv2.imread(os.path.join(homography_loc, "dst_0.jpg"))



gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
```

```
hessian_threshold = 42000
surf = cv2.xfeatures2d.SURF_create(hessian_threshold)
keypoints1, descriptors1 = surf.detectAndCompute(gray1, None)
img1_keypoints = cv2.drawKeypoints(gray1, keypoints1, None, (255, 0, 0), fla
plt.figure(figsize=(8, 8))
plt.subplot(1, 1, 1)
plt.imshow(img1_keypoints)
plt.title(f"Keypoints on dst_0.jpg({len(keypoints1)} features)")
plt.axis('off')
plt.tight_layout()
plt.show()
```

Keypoints on dst_0.jpg(28 features)

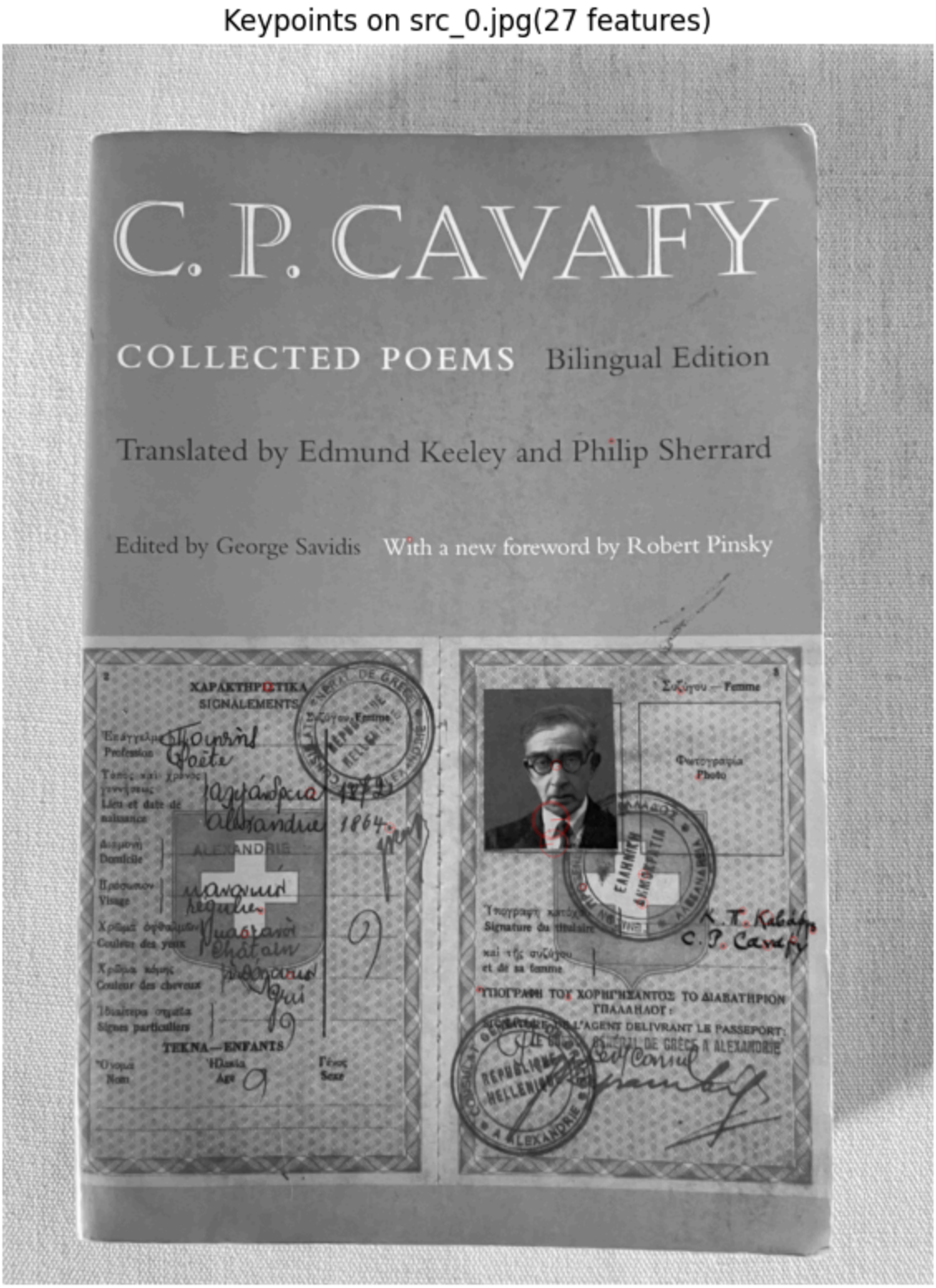
In [19]:
```
img2 = cv2.imread(os.path.join(homography_loc, "dst_1.jpg"))
gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

hessian_threshold = 25500
surf = cv2.xfeatures2d.SURF_create(hessian_threshold)
keypoints2, descriptors2 = surf.detectAndCompute(gray2, None)
img2_keypoints = cv2.drawKeypoints(gray2, keypoints2, None, (255, 0, 0), fla
```

```
plt.figure(figsize=(8, 8))

plt.subplot(1, 1,1)
plt.imshow(img2_keypoints)
plt.title(f"Keypoints on dst_1.jpg({len(keypoints2)} features)")
plt.axis('off')


plt.tight_layout()
plt.show()
```



In [34]:
```
img3 = cv2.imread(os.path.join(homography_loc, "src_0.jpg"))
gray3 = cv2.cvtColor(img3, cv2.COLOR_BGR2GRAY)
hessian_threshold = 16500
surf = cv2.xfeatures2d.SURF_create(hessian_threshold)
keypoints3, descriptors3 = surf.detectAndCompute(gray3, None)
img3_keypoints = cv2.drawKeypoints(gray3, keypoints3, None, (255, 0, 0), fla
plt.figure(figsize=(8, 8))
plt.subplot(1, 1, 1)
```

```
plt.imshow(img3_keypoints)
plt.title(f"Keypoints on src_0.jpg({len(keypoints3)} features)")
plt.axis('off')
plt.tight_layout()
plt.show()
```



In [38]:
```
img4 = cv2.imread(os.path.join(homography_loc, "src_1.jpg"))
gray4 = cv2.cvtColor(img4, cv2.COLOR_BGR2GRAY)
hessian_threshold = 46500
```

```
surf = cv2.xfeatures2d.SURF_create(hessian_threshold)
keypoints4, descriptors4 = surf.detectAndCompute(gray4, None)
img4_keypoints = cv2.drawKeypoints(gray4, keypoints4, None, (255, 0, 0), fla
plt.figure(figsize=(8, 8))
plt.subplot(1, 1, 1)
plt.imshow(img4_keypoints)
plt.title(f"Keypoints on src_1.jpg({len(keypoints4)} features)")
plt.axis('off')
plt.tight_layout()
plt.show()
```

Keypoints on src_1.jpg(28 features)

In [41]:
```
img5 = cv2.imread(os.path.join(homography_loc, "src_2.jpg"))
gray5 = cv2.cvtColor(img5, cv2.COLOR_BGR2GRAY)
hessian_threshold = 20000
surf = cv2.xfeatures2d.SURF_create(hessian_threshold)
keypoints5, descriptors5 = surf.detectAndCompute(gray5, None)
img5_keypoints = cv2.drawKeypoints(gray5, keypoints5, None, (255, 0, 0), fla
plt.figure(figsize=(8, 8))
plt.subplot(1, 1, 1)
```

```
plt.imshow(img5_keypoints)
plt.title(f"Keypoints on src_2.jpg({len(keypoints5)} features)")
plt.axis('off')
plt.tight_layout()
plt.show()
```

Feature Matching (10 pts)

Create a FLANN based matcher. Find matches among the descriptors you just detected between every pairs of src-dst images. Graphically show the top-20 matches found by the matcher (for each src-dst pair). You can follow the tutorial in https://docs.opencv.org/4.5.2/dc/dc3/tutorial_py_matcher.html

In [69]:
```
# # TODO
dst_imgs = [img1, img2]
dst_kps = [keypoints1, keypoints2]
dst_descs = [descriptors1, descriptors2]

# Group src images and their features (src images are img3, img4, and img5)
src_imgs = [img3, img4, img5]
src_kps = [keypoints3, keypoints4, keypoints5]
src_descs = [descriptors3, descriptors4, descriptors5]
# Create the FLANN based matcher
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=3)
search_params = dict(checks=500)
flann = cv2.FlannBasedMatcher(index_params, search_params)

# For every pair of dst and src images:
for i in range(len(dst_imgs)):
    for j in range(len(src_imgs)):
        # Use knnMatch with k=2 for the ratio test
        matches = flann.knnMatch(dst_descs[i], src_descs[j], k=2)

        # Apply Lowe's ratio test to filter good matches
        good_matches = []
        ratio_thresh = 0.95
        for m, n in matches:
            if m.distance < ratio_thresh * n.distance:
                good_matches.append(m)

        # Sort the good matches based on distance and take the top 20
        good_matches = sorted(good_matches, key=lambda x: x.distance)
        top_matches = good_matches[:20]

        # Draw matches using a bright, contrasting match color (red in this
        matched_img = cv2.drawMatches(dst_imgs[i], dst_kps[i],
                                      src_imgs[j], src_kps[j],
                                      top_matches, None,
                                      matchColor=(0, 0, 255),
                                      flags=cv2.DrawMatchesFlags_NOT_DRAW_SI

        # Optionally, reinforce the match lines by drawing them manually wit
        # Get the width of the destination image to compute the offset for s
        offset = dst_imgs[i].shape[1]
        for match in top_matches:
            # Get the coordinates of the keypoints from the destination imag
            pt_dst = tuple(map(int, dst_kps[i][match.queryIdx].pt))
            # ...and from the src image (remember to add the offset)
            pt_src = tuple(map(int, src_kps[j][match.trainIdx].pt))
            pt_src_adjusted = (pt_src[0] + offset, pt_src[1])
            # Redraw the matching line with a thicker stroke
```

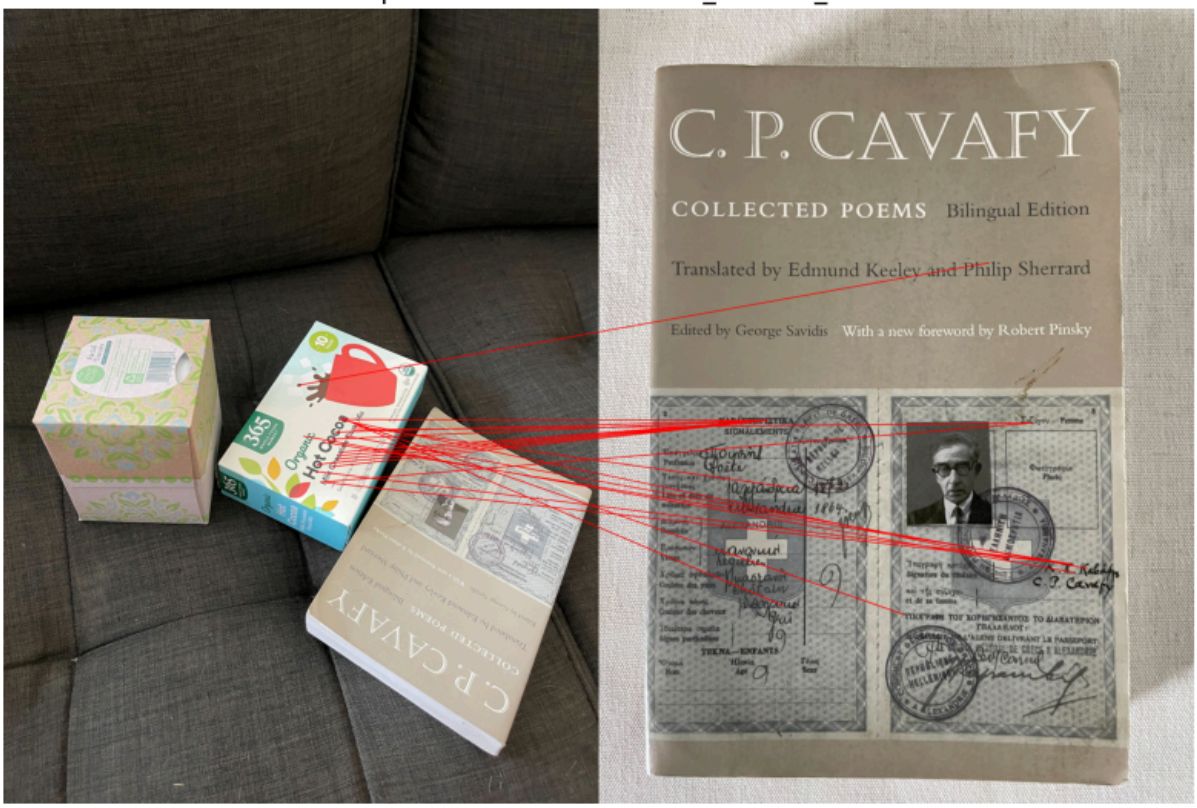
```
            cv2.line(matched_img, pt_dst, pt_src_adjusted, (0, 0, 255), thic

        # Display the result using matplotlib
        plt.figure(figsize=(10, 8))
        plt.imshow(cv2.cvtColor(matched_img, cv2.COLOR_BGR2RGB))
        plt.title(f"Top-20 FLANN Matches: dst_{i} vs src_{j}")
        plt.axis('off')
        plt.show()
```

Top-20 FLANN Matches: dst_0 vs src_0

Top-20 FLANN Matches: dst_0 vs src_1

Top-20 FLANN Matches: dst_0 vs src_2

Top-20 FLANN Matches: dst_1 vs src_0



Top-20 FLANN Matches: dst_1 vs src_1

Compute Homography (10 pts)

Compute the homography using RANSAC. Print out the homography matrix. Transform the four corners of the source image using the homography and display the transformed rectangle on the destination image. You can follow the tutorial in https://docs.opencv.org/4.5.2/d1/de0/tutorial_py_feature_homography.html

In [ ]:
```
import os
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Define the folder with your homography images
homography_loc = "/content/drive/MyDrive/1acv/csci677_2025_assignment1_data/

# Load destination and source images (in color)
dst_img1 = cv2.imread(os.path.join(homography_loc, "dst_0.jpg"))
dst_img2 = cv2.imread(os.path.join(homography_loc, "dst_1.jpg"))
src_img1 = cv2.imread(os.path.join(homography_loc, "src_0.jpg"))
src_img2 = cv2.imread(os.path.join(homography_loc, "src_1.jpg"))
src_img3 = cv2.imread(os.path.join(homography_loc, "src_2.jpg"))

# Convert images to grayscale for feature detection
gray_dst1 = cv2.cvtColor(dst_img1, cv2.COLOR_BGR2GRAY)
gray_dst2 = cv2.cvtColor(dst_img2, cv2.COLOR_BGR2GRAY)
gray_src1 = cv2.cvtColor(src_img1, cv2.COLOR_BGR2GRAY)
gray_src2 = cv2.cvtColor(src_img2, cv2.COLOR_BGR2GRAY)
```

```python
gray_src3 = cv2.cvtColor(src_img3, cv2.COLOR_BGR2GRAY)

# Create SURF detector instance (ensure opencv-contrib is installed)
hessian_threshold = 100  # Lowering threshold to detect more keypoints
surf = cv2.xfeatures2d.SURF_create(hessian_threshold)

# Detect keypoints and compute descriptors
dst_kps1, dst_desc1 = surf.detectAndCompute(gray_dst1, None)
dst_kps2, dst_desc2 = surf.detectAndCompute(gray_dst2, None)
src_kps1, src_desc1 = surf.detectAndCompute(gray_src1, None)
src_kps2, src_desc2 = surf.detectAndCompute(gray_src2, None)
src_kps3, src_desc3 = surf.detectAndCompute(gray_src3, None)

# Group destination and source images along with their features
dst_imgs = [dst_img1, dst_img2]
dst_kps  = [dst_kps1, dst_kps2]
dst_desc = [dst_desc1, dst_desc2]

src_imgs = [src_img1, src_img2, src_img3]
src_kps  = [src_kps1, src_kps2, src_kps3]
src_desc = [src_desc1, src_desc2, src_desc3]

# Setup BFMatcher for SURF descriptors (float descriptors; NORM_L2)
bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=False)

# # Set minimum match count to consider a successful homography computation
# MIN_MATCH_COUNT = 10

for i in range(len(dst_imgs)):
    for j in range(len(src_imgs)):
        # k-NN matching (k=2) with Lowe's ratio test
        matches = bf.knnMatch(src_desc[j], dst_desc[i], k=2)
        good_matches = []
        ratio_thresh = 0.7
        for m, n in matches:
            if m.distance < ratio_thresh * n.distance:
                good_matches.append(m)

        print(f"Dst image {i} vs. Src image {j}: {len(good_matches)} good ma

        # Check if there are enough good matches
        # if len(good_matches) < MIN_MATCH_COUNT:
        #     print(f"Not enough good matches between Dst image {i} and Src
        #     continue

        # Build point arrays from the good matches
        src_pts = np.float32([src_kps[j][m.queryIdx].pt for m in good_matche
        dst_pts = np.float32([dst_kps[i][m.trainIdx].pt for m in good_matche

        # Compute homography using RANSAC
        H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 3.0)
        # if H is None:
        #     print(f"Homography could not be computed for Dst image {i} vs.
        #     continue

        print(f"Homography for Dst {i} vs. Src {j}:\n{H}\n")
```

```
        # Transform the four corners of the source image using the computed
        h_src, w_src = src_imgs[j].shape[:2]
        corners = np.float32([[0, 0], [w_src, 0], [w_src, h_src], [0, h_src]
        transformed_corners = cv2.perspectiveTransform(corners, H)

        # Draw the transformed quadrilateral (red square) on a copy of the d
        dst_with_rect = dst_imgs[i].copy()
        cv2.polylines(dst_with_rect, [np.int32(transformed_corners)],
                      isClosed=True, color=(0, 0, 255), thickness=3)

        # --- Create composite image with src on left, dst with square on ri
        h_src_img, w_src_img = src_imgs[j].shape[:2]
        h_dst_img, w_dst_img = dst_with_rect.shape[:2]
        composite_height = max(h_src_img, h_dst_img)
        composite_width  = w_src_img + w_dst_img

        composite = np.zeros((composite_height, composite_width, 3), dtype=r
        composite[:h_src_img, :w_src_img] = src_imgs[j]
        composite[:h_dst_img, w_src_img:w_src_img+w_dst_img] = dst_with_rect

        # Compute centers: source image center and the center of the transfo
        src_center = (w_src_img // 2, h_src_img // 2)
        transformed_corners_reshaped = transformed_corners.reshape(4, 2)
        center_poly = np.mean(transformed_corners_reshaped, axis=0)  # cente

        # Adjust dst center for composite image (dst placed at x offset = w_
        dst_center = (int(w_src_img + center_poly[0]), int(center_poly[1]))

        # Draw a green line from src center to dst polygon center
        cv2.line(composite, src_center, dst_center, (0, 255, 0), thickness=2

        # Display the composite image with matplotlib
        plt.figure(figsize=(10, 10))
        plt.imshow(cv2.cvtColor(composite, cv2.COLOR_BGR2RGB))
        plt.title(f"Src image {j} and Dst image {i} with transformed square
        plt.axis('off')
        plt.show()
```

```
Dst image 0 vs. Src image 0: 595 good matches
Homography for Dst 0 vs. Src 0:
[[ 4.91163288e-01  2.94304385e-01  1.34389619e+02]
 [-1.32451654e-01  1.75800086e-01  2.13300225e+03]
 [ 7.53499584e-05 -5.93090034e-05  1.00000000e+00]]
```

Src image 0 and Dst image 0 with transformed square and connecting line


```
Dst image 0 vs. Src image 1: 440 good matches
Homography for Dst 0 vs. Src 1:
[[-2.09377029e-01 -3.80926127e-01  2.18849469e+03]
 [ 5.75814001e-02  8.38613006e-03  4.75009489e+02]
 [-1.75362560e-04 -1.01663600e-05  1.00000000e+00]]
```

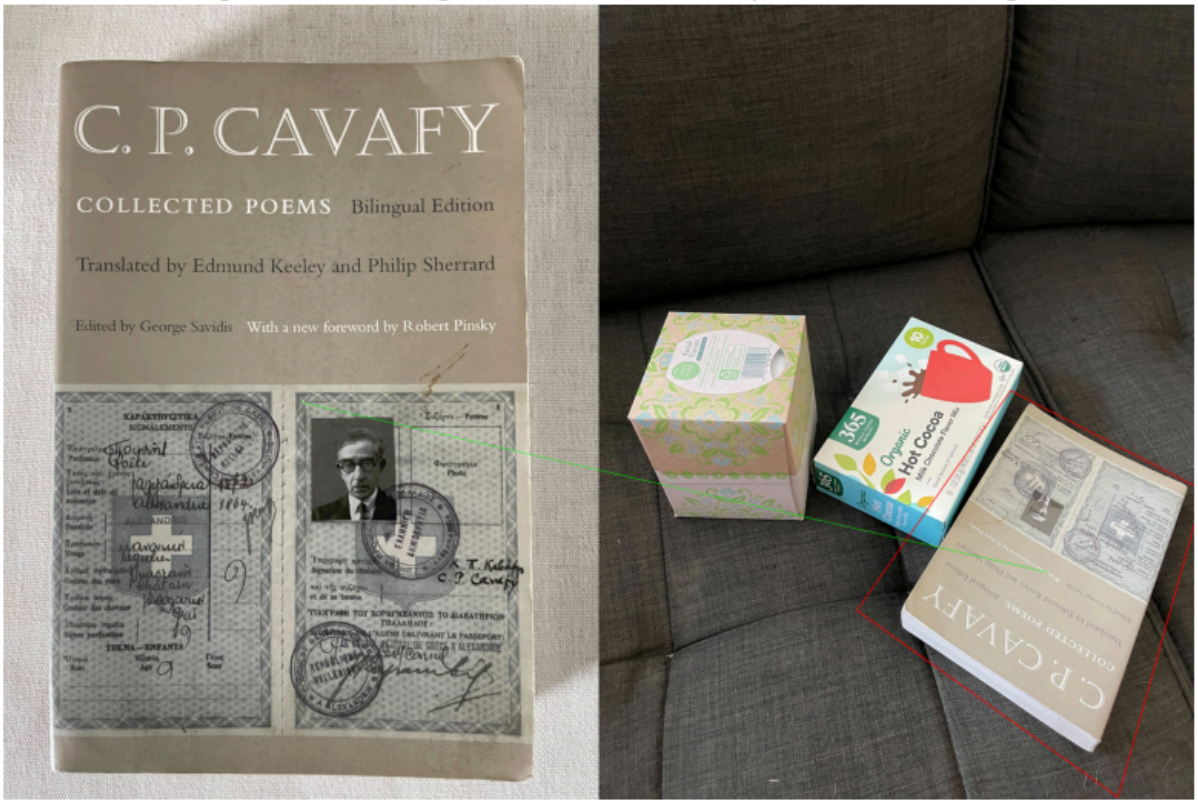
Src image 1 and Dst image 0 with transformed square and connecting line


```
Dst image 0 vs. Src image 2: 151 good matches
Homography for Dst 0 vs. Src 2:
[[8.06976273e-01 2.05032888e-01 6.88433536e+02]
 [2.75746410e-01 7.01420599e-01 1.46070441e+03]
 [1.01535625e-04 1.52164126e-04 1.00000000e+00]]
```

Src image 2 and Dst image 0 with transformed square and connecting line


```
Dst image 1 vs. Src image 0: 531 good matches
Homography for Dst 1 vs. Src 0:
[[-3.63938489e-01  4.78707352e-01  2.63725030e+03]
 [-1.38684419e-01 -8.82219480e-02  4.01919092e+03]
 [ 5.67972314e-05  1.20502146e-04  1.00000000e+00]]
```

Src image 0 and Dst image 1 with transformed square and connecting line




```
Dst image 1 vs. Src image 1: 401 good matches
Homography for Dst 1 vs. Src 1:
[[ 2.10187156e-01 -1.81923731e-01  1.50112391e+03]
 [ 5.56558555e-02  1.08924723e-01  1.48643959e+03]
 [-2.36589350e-05 -4.33998497e-05  1.00000000e+00]]
```

Src image 1 and Dst image 1 with transformed square and connecting line


```
Dst image 1 vs. Src image 2: 143 good matches
Homography for Dst 1 vs. Src 2:
[[ 2.21904115e-01 -6.79657867e-01  8.09338274e+02]
 [ 4.96113170e-01 -1.51926289e+00  1.80901118e+03]
 [ 2.74196629e-04 -8.39786154e-04  1.00000000e+00]]
```

Src image 2 and Dst image 1 with transformed square and connecting line

Observation (5 pts)

Write down your observations regarding the results you obtained throughout the `Homography` section.

Observations for Homography

1. Feature Detection: SURF detected features on the images, but the Hessian threshold has to be carefully tuned. A higher Hessian threshold results in fewer keypoints, potentially leading to less accurate matches and homography calculation, but also a faster computation. A lower threshold, leads to more keypoints, but could result in a less accurate result.

2. Feature Matching: The FLANN matcher identified correspondences between the source and destination images. The top 20 matches (in terms of distance) were visualized.

3. Homography Computation: The homography matrix was computed using RANSAC, which is a robust method to handle outliers in the matched keypoints. The accuracy of the homography is influenced by the quality of the matches. A higher number of good matches with correct correspondence usually lead to better homography results.

4. Corner Transformation: The computed homography was applied to transform the corners of the source image, which is shown on top of the destination image. A visually well-aligned rectangle suggests a good homography estimation. It's important to check if the transformed corners precisely align with the corresponding region in the destination image. Only the last image was not able to get a square around it as there are not a lot of detectable features in that image.

Panorama (30 pts)

Data Preparation

Please download the `data.zip` file from the link, and extract it to a location of your choice. For this part of the assignment, use the images in the `panorama` folder.

Compute Homography (10 pts)

Here we are computing homography again, but once every two consecutive images. To do this, you need to first import the images. Then you pick a feature detector (not necessarily SIFT) and detect features. Then you pick a feature matcher (not necessarily brute-force) and find matches between every two consecutive images. Then you compute homography and store them. Below we write a skeleton for you, but you needn't follow it.

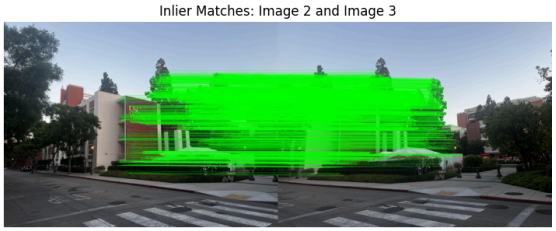
In [ ]:
```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

def show_image(img, figsize=(18, 12)):
    plt.figure(figsize=figsize)
    plt.imshow(img)
    plt.axis('off')
    plt.show()

# Load images using cv2.imread (BGR format)
left_img  = cv2.imread("/content/drive/MyDrive/1acv/csci677_2025_assignment1
cent_img  = cv2.imread("/content/drive/MyDrive/1acv/csci677_2025_assignment1
right_img = cv2.imread("/content/drive/MyDrive/1acv/csci677_2025_assignment1


    # Convert each image from BGR to RGB
left_img  = cv2.cvtColor(left_img, cv2.COLOR_BGR2RGB)
cent_img  = cv2.cvtColor(cent_img, cv2.COLOR_BGR2RGB)
right_img = cv2.cvtColor(right_img, cv2.COLOR_BGR2RGB)

# Concatenate images along axis=1 (i.e., horizontally)
combined_img = np.concatenate([left_img, cent_img, right_img], axis=1)

# Display the concatenated image
show_image(combined_img)
```

In [ ]:
```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

```python
def load_image(path):
    img = cv2.imread(path)
    if img is None:
        print("Error loading image at:", path)
    return img

def compute_homography_and_inliers(img1, img2, ratio=0.75, ransac_thresh=5.0
    # Convert images to grayscale for keypoint detection
    gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

    # Initialize SIFT detector and extract keypoints and descriptors
    sift = cv2.SIFT_create()
    kp1, des1 = sift.detectAndCompute(gray1, None)
    kp2, des2 = sift.detectAndCompute(gray2, None)

    # Use BFMatcher and perform knn matching with k=2
    bf = cv2.BFMatcher()
    knn_matches = bf.knnMatch(des1, des2, k=2)

    # Apply Lowe's ratio test to filter out ambiguous matches
    good_matches = []
    for m, n in knn_matches:
        if m.distance < ratio * n.distance:
            good_matches.append(m)

    # At least 4 good matches are needed to compute homography.
    if len(good_matches) >= 4:
        src_pts = np.float32([kp1[m.queryIdx].pt for m in good_matches]).res
        dst_pts = np.float32([kp2[m.trainIdx].pt for m in good_matches]).res
        H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, ransac_th
    else:
        H, mask = None, None

    return kp1, kp2, good_matches, H, mask

def draw_inlier_matches(img1, img2, kp1, kp2, matches, inlier_mask):
    # inlier_mask is returned by findHomography as an array of 0s and 1s.
    draw_params = {
        "matchColor": (0, 255, 0),  # green lines for inliers
        "singlePointColor": None,
        "matchesMask": inlier_mask.ravel().tolist(),  # mask of inliers
        "flags": 2
    }
    return cv2.drawMatches(img1, kp1, img2, kp2, matches, None, **draw_param

def main():
    # Update these paths to point to your images
    path1 = "/content/drive/MyDrive/1acv/csci677_2025_assignment1_data/panor
    path2 = "/content/drive/MyDrive/1acv/csci677_2025_assignment1_data/panor
    path3 = "/content/drive/MyDrive/1acv/csci677_2025_assignment1_data/panor

    img1 = load_image(path1)
    img2 = load_image(path2)
    img3 = load_image(path3)
```

```
    if (img1 is None) or (img2 is None) or (img3 is None):
        print("One of the images did not load. Check your file paths.")
        return

    # Compute homography and inliers between image 1 and image 2
    kp1_12, kp2_12, good_matches_12, H12, mask12 = compute_homography_and_ir
    if H12 is not None:
        print("Homography between Image 1 and Image 2:")
        print(H12)
        print("Number of inliers:", np.sum(mask12))
        img_matches_12 = draw_inlier_matches(img1, img2, kp1_12, kp2_12, goo
    else:
        print("Not enough matches to compute homography between Image 1 and
        return

    # Compute homography and inliers between image 2 and image 3
    kp2_23, kp3_23, good_matches_23, H23, mask23 = compute_homography_and_ir
    if H23 is not None:
        print("Homography between Image 2 and Image 3:")
        print(H23)
        print("Number of inliers:", np.sum(mask23))
        img_matches_23 = draw_inlier_matches(img2, img3, kp2_23, kp3_23, goo
    else:
        print("Not enough matches to compute homography between Image 2 and
        return

    # Display the inlier matches using matplotlib
    plt.figure(figsize=(20, 10))

    plt.subplot(1, 2, 1)
    plt.title("Inlier Matches: Image 1 and Image 2")
    plt.imshow(cv2.cvtColor(img_matches_12, cv2.COLOR_BGR2RGB))
    plt.axis("off")

    plt.subplot(1, 2, 2)
    plt.title("Inlier Matches: Image 2 and Image 3")
    plt.imshow(cv2.cvtColor(img_matches_23, cv2.COLOR_BGR2RGB))
    plt.axis("off")

    plt.show()


if __name__ == "__main__":
    main()
```

```
Homography between Image 1 and Image 2:
[[ 1.47528093e+00 -4.62819233e-02 -1.33342631e+03]
 [ 2.25302869e-01  1.29650791e+00 -5.99816380e+02]
 [ 1.14821958e-04  6.77674824e-06  1.00000000e+00]]
Number of inliers: 5902
Homography between Image 2 and Image 3:
[[ 1.58950028e+00 -6.02160160e-02 -1.60347389e+03]
 [ 2.65764410e-01  1.33575101e+00 -6.25880950e+02]
 [ 1.45378693e-04 -6.71883688e-06  1.00000000e+00]]
Number of inliers: 3556
```



Stitch Panorama (15 pts)

Now we can stitch those images to compose a panorama. You need to select an image as an anchor and transform other images onto this anchor image. The transformation between any image and this anchor image is the composition of a series of homography. You should compute the transformations and map all other images onto the anchor image. You can explore other ways to define an anchor. Then you need to blend these image. A possible way is to just take the maximum of the pixel values, but you are encouraged to explore other blending methods (extra points +2~5 pts). After you obtained your panorama, display it along with some intermediate results including feature matches and transformed images. We attached an example in the folder `panorama_output`. Below we provide the code to compute the size of a rectangle that covers all transformed images, but you needn't follow it.

In [ ]:
```python
import cv2
import numpy as np
from matplotlib import pyplot as plt
from google.colab.patches import cv2_imshow
from PIL import Image

# --- Assume genSIFTMatchPairs and RANSAC are defined elsewhere ---

def compute_output_canvas(imgs, homographies, pad=(0,0)):
    """
    Computes the overall canvas size needed to fit all warped images.
    imgs: list of images.
    homographies: list of homography matrices for each image that maps that
    pad: additional padding (pad_x, pad_y)
    Returns: (min_x, min_y, max_x, max_y)
    """
    min_x = float('inf')
    min_y = float('inf')
    max_x = float('-inf')
    max_y = float('-inf')

    for i, img in enumerate(imgs):
        h, w = img.shape[:2]
        # Define corners of the image.
        corners = np.array([[0, 0], [w, 0], [w, h], [0, h]], dtype=np.float3
        warped_corners = cv2.perspectiveTransform(corners, homographies[i])
        warped_corners = warped_corners.reshape(-1, 2)
        min_x = min(min_x, warped_corners[:, 0].min())
        min_y = min(min_y, warped_corners[:, 1].min())
```

```
        max_x = max(max_x, warped_corners[:, 0].max())
        max_y = max(max_y, warped_corners[:, 1].max())

    # Add optional padding
    pad_x, pad_y = pad
    min_x -= pad_x
    min_y -= pad_y
    max_x += pad_x
    max_y += pad_y
    return min_x, min_y, max_x, max_y

def stitch_img(imgs):
    '''
    Stitch three images into a panorama using the second image as the anchor
    Input:
      imgs: list of three images [center_img, left_img, right_img]
    Output:
      stitched_img: the final stitched panorama.
    '''
    # Use the center image as the anchor.
    anchor_img = imgs[0]  # imgs[0] is the center image.
    left_img = imgs[1]
    right_img = imgs[2]

    # Compute homographies using your SIFT matching routines.
    # For left_img to anchor:
    pts1, pts2, matches1to2, kp1, kp2 = genSIFTMatchPairs(left_img, anchor_i
    inliers_idx, H_left = RANSAC(pts1, pts2, 500, 20)

    # For right_img to anchor:
    pts1_r, pts2_r, matches1to2_r, kp1_r, kp2_r = genSIFTMatchPairs(right_im
    inliers_idx_r, H_right = RANSAC(pts1_r, pts2_r, 500, 20)

    print("Homography for left->anchor:")
    print(H_left)
    print("Homography for right->anchor:")
    print(H_right)

    # Instead of warping onto a fixed padded anchor, we'll compute a common
    # First, define the homographies for each image in the anchor coordinate
    # For the anchor image, use identity.
    H_anchor = np.eye(3, dtype=np.float32)

    # For left and right images, H_left and H_right are assumed to map them
    homographies = [H_anchor, H_left, H_right]

    # Compute the overall canvas bounds:
    min_x, min_y, max_x, max_y = compute_output_canvas(imgs, homographies, p
    output_width = int(np.ceil(max_x - min_x))
    output_height = int(np.ceil(max_y - min_y))
    print("Output canvas size:", output_width, "x", output_height)

    # Create a translation matrix to shift everything into positive coordina
    translation = np.array([[1, 0, -min_x],
                            [0, 1, -min_y],
                            [0, 0, 1]], dtype=np.float32)
```

```
    # Warp each image onto the common canvas.
    warped_left = cv2.warpPerspective(left_img, translation @ H_left, (outpu
    warped_anchor = cv2.warpPerspective(anchor_img, translation @ H_anchor,
    warped_right = cv2.warpPerspective(right_img, translation @ H_right, (ou

    # For blending, we can use a simple weighted average in overlapping area
    # Here we simply compute a pixel-wise maximum for demonstration.
    # (You might want to use more advanced blending, e.g. multi-band blendin
    panorama = np.maximum(np.maximum(warped_anchor, warped_left), warped_rig

    return panorama


center_img = cv2.imread("/content/drive/MyDrive/1acv/csci677_2025_assignment
left_img   = cv2.imread("/content/drive/MyDrive/1acv/csci677_2025_assignment
right_img  = cv2.imread("/content/drive/MyDrive/1acv/csci677_2025_assignment

final_stitch = stitch_img([center_img, left_img, right_img])

# Optionally crop the final result if needed.
# (Cropping parameters might need adjustment based on your results.)
final_stitch_cropped = final_stitch[100:final_stitch.shape[0]-100, 100:final

plt.figure(figsize=(16,12))
plt.imshow(cv2.cvtColor(final_stitch_cropped.astype("uint8"), cv2.COLOR_BGR2
plt.title("Final Stitched Panorama")
plt.axis("off")
plt.show()
```

```
Homography for left->anchor:
[[ 1.47789158e+00 -4.70845822e-02 -1.32558847e+03]
 [ 2.31030541e-01  1.29695537e+00 -6.07837122e+02]
 [ 1.17805780e-04  5.16717209e-06  1.00000000e+00]]
Homography for right->anchor:
[[ 6.28417756e-01  3.70139950e-02  1.02156049e+03]
 [-1.65805475e-01  8.66952026e-01  2.55651983e+02]
 [-9.08929867e-05  5.89727055e-06  1.00000000e+00]]
Output canvas size: 7876 x 4843
```

Final Stitched Panorama

Second Mathod: Grid Blending

A grid of destination (canvas) coordinates is generated, and each point is mapped back into the source image using the inverse homography matrix. Nearest–neighbor interpolation assigns pixel values from the source image to the output, effectively "warping" the image onto a common canvas.

In [ ]:
```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

def load_image(path):
    img = cv2.imread(path)
    if img is None:
        raise IOError(f"Could not load image at: {path}")
    return img

def compute_homography_and_inliers(img1, img2, ratio=0.75, ransac_thresh=5.0
    # Convert to grayscale for SIFT detection
    gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
    sift = cv2.SIFT_create()
    kp1, des1 = sift.detectAndCompute(gray1, None)
    kp2, des2 = sift.detectAndCompute(gray2, None)

    # Use BFMatcher with k-NN
    bf = cv2.BFMatcher()
    knn_matches = bf.knnMatch(des1, des2, k=2)

    good_matches = []
    for m, n in knn_matches:
```

```
        if m.distance < ratio * n.distance:
            good_matches.append(m)

    if len(good_matches) >= 4:
        src_pts = np.float32([kp1[m.queryIdx].pt for m in good_matches]).res
        dst_pts = np.float32([kp2[m.trainIdx].pt for m in good_matches]).res
        H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, ransac_th
    else:
        H, mask = None, None
    return kp1, kp2, good_matches, H, mask

def draw_inlier_matches(img1, img2, kp1, kp2, matches, mask):
    # Draw only the matches flagged as inliers by RANSAC (mask)
    draw_params = dict(matchColor=(0, 255, 0),
                       singlePointColor=None,
                       matchesMask=mask.ravel().tolist(),
                       flags=2)
    return cv2.drawMatches(img1, kp1, img2, kp2, matches, None, **draw_param

def manual_warp(src, M, output_shape):
    """
    Manually warp the source image using inverse mapping.
    M is a 3x3 transformation matrix.
    output_shape is (height, width).
    """
    # Compute inverse of the mapping
    H_inv = np.linalg.inv(M)
    out_h, out_w = output_shape
    # Create grid of destination coordinates
    xv, yv = np.meshgrid(np.arange(out_w), np.arange(out_h))
    ones = np.ones_like(xv)
    dest_coords = np.stack([xv, yv, ones], axis=-1).reshape(-1, 3).T  # shap

    # Map destination pixels back to source coordinates
    src_coords = H_inv @ dest_coords
    src_coords /= src_coords[2, :]
    src_x = src_coords[0, :].reshape(out_h, out_w)
    src_y = src_coords[1, :].reshape(out_h, out_w)

    # Nearest-neighbor interpolation
    src_x_int = np.round(src_x).astype(int)
    src_y_int = np.round(src_y).astype(int)

    output = np.zeros((out_h, out_w, src.shape[2]), dtype=src.dtype)
    valid = (src_x_int >= 0) & (src_x_int < src.shape[1]) & (src_y_int >= 0)
    output[valid] = src[src_y_int[valid], src_x_int[valid]]
    return output

def get_transformed_corners(img, M):
    h, w = img.shape[:2]
    # Four corners of the image
    corners = np.array([[0, 0], [0, h], [w, h], [w, 0]], dtype=np.float32)
    # Convert to homogeneous coordinates
    corners_hom = np.hstack([corners, np.ones((4, 1), dtype=np.float32)])  #
    transformed = (M @ corners_hom.T).T  # shape (4,3)
    transformed /= transformed[:, 2:3]
```

```
    return transformed[:, :2]

def compute_canvas_bounds(left, center, right, M_left, M_right):
    # Compute transformed corners using the provided transformation matrices
    corners_left = get_transformed_corners(left, M_left)
    corners_center = get_transformed_corners(center, np.eye(3))
    corners_right = get_transformed_corners(right, M_right)
    all_corners = np.concatenate([corners_left, corners_center, corners_righ
    min_xy = np.floor(all_corners.min(axis=0)).astype(int)
    max_xy = np.ceil(all_corners.max(axis=0)).astype(int)
    return min_xy, max_xy

def stitch_panorama_manual(left, center, right, H_left, H_right):
    # H_left: left -> center; for right, we want mapping from right -> cente
    M_left = H_left
    M_right = np.linalg.inv(H_right)

    # Compute overall canvas bounds using the transformed corners.
    min_xy, max_xy = compute_canvas_bounds(left, center, right, M_left, M_ri
    T = np.array([[1, 0, -min_xy[0]],
                  [0, 1, -min_xy[1]],
                  [0, 0, 1]], dtype=np.float32)

    # New transformation matrices incorporate the translation offset.
    M_left_t   = T @ M_left
    M_center_t = T  # center image; only translated.
    M_right_t  = T @ M_right

    canvas_width  = max_xy[0] - min_xy[0]
    canvas_height = max_xy[1] - min_xy[1]
    output_shape = (canvas_height, canvas_width)

    # Use our manual_warp function to warp each image.
    warped_left   = manual_warp(left, M_left_t, output_shape)
    warped_center = manual_warp(center, M_center_t, output_shape)
    warped_right  = manual_warp(right, M_right_t, output_shape)

    # Simple overlay: start with center as the base.
    panorama = warped_center.copy()
    mask_left  = (warped_left.sum(axis=2) > 0)
    mask_right = (warped_right.sum(axis=2) > 0)
    panorama[mask_left]  = warped_left[mask_left]
    panorama[mask_right] = warped_right[mask_right]

    return panorama

def main():
    # Update these paths with your image file locations.
    path_left   = "/content/drive/MyDrive/1acv/csci677_2025_assignment1_data
    path_center = "/content/drive/MyDrive/1acv/csci677_2025_assignment1_data
    path_right  = "/content/drive/MyDrive/1acv/csci677_2025_assignment1_data

    left_img   = load_image(path_left)
    center_img = load_image(path_center)
    right_img  = load_image(path_right)
```

```
    # Compute homography between left and center images.
    kp_left, kp_center, matches_lc, H_lc, mask_lc = compute_homography_and_i
    if H_lc is None:
        print("Not enough matches between left and center images.")
        return
    img_matches_lc = draw_inlier_matches(left_img, center_img, kp_left, kp_c
    # plt.figure(figsize=(12, 6))
    # plt.title("Inlier Matches: Left - Center")
    # plt.imshow(cv2.cvtColor(img_matches_lc, cv2.COLOR_BGR2RGB))
    # plt.axis("off")
    # plt.show()

    # Compute homography between center and right images (center -> right ma
    kp_center2, kp_right, matches_cr, H_cr, mask_cr = compute_homography_and
    if H_cr is None:
        print("Not enough matches between center and right images.")
        return
    img_matches_cr = draw_inlier_matches(center_img, right_img, kp_center2,
    # plt.figure(figsize=(12, 6))
    # plt.title("Inlier Matches: Center - Right")
    # plt.imshow(cv2.cvtColor(img_matches_cr, cv2.COLOR_BGR2RGB))
    # plt.axis("off")
    # plt.show()

    # Create panorama using the manual warping method.
    panorama = stitch_panorama_manual(left_img, center_img, right_img, H_lc,
    plt.figure(figsize=(20, 10))
    plt.title("Panorama (Manual Warping without cv2.warpPerspective)")
    plt.imshow(cv2.cvtColor(panorama, cv2.COLOR_BGR2RGB))
    plt.axis("off")
    plt.show()

if __name__ == "__main__":
    main()
```

Panorama (Manual Warping without cv2.warpPerspective)

Observation (5 pts)

Write down your observations regarding the results you obtained throughout the `Panorama` section.

TODO: write down your observations

Observations for the Panorama Section

1. Homography Computation:
    - The homography matrices computed between image pairs represent the geometric transformations needed to align the images.
    - Higher inlier counts generally indicate better alignment and a more reliable transformation.
    - The quality of the matches and the estimated homographies directly influence the final stitched panorama. Poor matches or incorrect homographies will lead to misaligned images and a distorted panorama.
2. Image Stitching:
    - The stitching process involves warping each image onto a common canvas based on the computed homographies. The code uses the second image (center_img) as an anchor.
    - Padding and cropping: Padding is added to the canvas during the warping stage to prevent artifacts at the borders of the images.
3. Stitching Results:
    - The stitched panorama shows distortion at very few places and the complete picture is visible.