Program Architecture

Initialization and Setup:

* Load necessary libraries (cv2 for OpenCV and numpy for numerical operations).
* Define the list of image file names to be stitched.
* Set the dimensions of the output mosaic image (MOSAIC\_WIDTH and MOSAIC\_HEIGHT).

Main Function Workflow:

* Feature Detection and Matching: Initialize the feature detector (ORB) and the descriptor matcher (BFMatcher).
* First Image Processing: Read the first image, hardcode the corner points (for development), and define the orthophoto coordinates.
* Homography Calculation: Compute the homography matrix to warp the first image onto the orthophoto.
* Mosaic Initialization: Create the initial mosaic by warping the first image.

Iterative Processing of Remaining Images

For each subsequent image:

* Read the current image.
* Detect and compute features in the current and previous images.
* Match the descriptors and filter good matches.
* Compute the homography matrix between the current and previous images.
* Update the homography matrix to transform the current image onto the mosaic.
* Warp the current image and blend it with the mosaic.
* Update the previous image and homography matrix for the next iteration.

Finalization:

* Save the final mosaic image.
* Prompt the user to close all OpenCV windows.
* Properly release resources and close all windows.

Methods and Techniques Used

**Feature Detection (ORB)**

ORB (Oriented FAST and Rotated BRIEF) is used to detect and compute keypoints and descriptors in the images. This helps in finding unique points that can be matched between different images.

**Feature Matching (BFMatcher)**

BFMatcher (Brute-Force Matcher) is used to find the best matches between descriptors of keypoints in different images. K-nearest neighbors matching (knnMatch) with a ratio test is used to filter out poor matches.

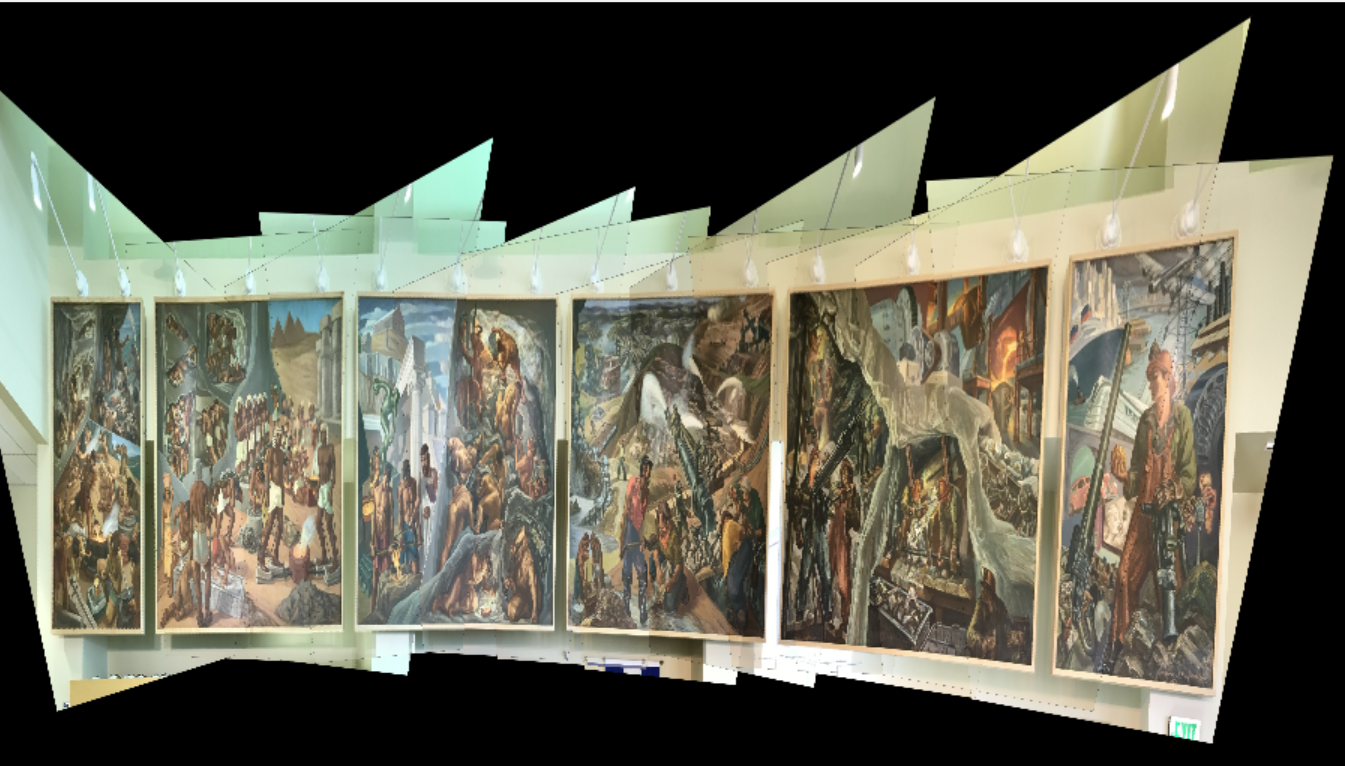
**Homography Estimation**

Homography is a transformation that maps points from one plane to another. cv2.findHomography with RANSAC (Random Sample Consensus) is used to compute the homography matrix that aligns one image with another. This is essential for warping images to align them in a mosaic.

**Image Warping and Blending**

cv2.warpPerspective is used to apply the homography matrix to warp images. The fuse\_color\_images function blends the warped images together, handling overlaps by averaging pixel values where both images contribute.

***Result:***



***Code:***

# -\*- coding: utf-8 -\*-

"""

@author: zezva

"""

import cv2

import numpy as np

fileNames = [

    'mural01.jpg', 'mural02.jpg', 'mural03.jpg',

    'mural04.jpg', 'mural05.jpg', 'mural06.jpg',

    'mural07.jpg', 'mural08.jpg', 'mural09.jpg',

    'mural10.jpg', 'mural11.jpg', 'mural12.jpg',

]

MOSAIC\_WIDTH = 7000

MOSAIC\_HEIGHT = 1300

def main():

    print("Stitch together images of a planar object")

    # Initialize the ORB feature detector

    detector = cv2.ORB\_create(nfeatures=2000, nlevels=8, firstLevel=0, patchSize=31, edgeThreshold=31)

    # Initialize the BFMatcher

    matcher = cv2.BFMatcher(cv2.NORM\_HAMMING, crossCheck=False)

    # Read the first image

    bgr\_first = cv2.imread(fileNames[0])

    # Hardcode the corners for the first image

    corners = [(156, 123), (377, 73), (299, 495), (32, 475)]

    print("Ok, here are the corner locations:", corners)

    # Define the actual size of the leftmost mural panel

    leftmost\_mural\_width\_meters = 1.593

    leftmost\_mural\_height\_meters = 2.178

    # Define the scale of the output orthophoto

    pixel\_size\_cm = 0.5

    pixel\_size\_m = pixel\_size\_cm / 100

    pixels\_per\_meter = 1 / pixel\_size\_m

    # Set the size of the leftmost mural in the output orthophoto

    ortho\_width = int(round(leftmost\_mural\_width\_meters \* pixels\_per\_meter))

    ortho\_height = int(round(leftmost\_mural\_height\_meters \* pixels\_per\_meter))

    # Set the offset for the first panel in the output image

    offset\_x\_pixels = 200

    offset\_y\_pixels = 400

    ortho\_corners = np.array([

        [0 + offset\_x\_pixels, 0 + offset\_y\_pixels],

        [ortho\_width + offset\_x\_pixels, 0 + offset\_y\_pixels],

        [ortho\_width + offset\_x\_pixels, ortho\_height + offset\_y\_pixels],

        [0 + offset\_x\_pixels, ortho\_height + offset\_y\_pixels]

    ])

    # Find the homography that maps the first image to the orthophoto coordinates

    src\_pts = np.array(corners)

    H, \_ = cv2.findHomography(srcPoints=src\_pts, dstPoints=ortho\_corners)

    # Warp the first image to create the initial mosaic

    bgr\_mosaic = cv2.warpPerspective(bgr\_first, H, (MOSAIC\_WIDTH, MOSAIC\_HEIGHT))

    cv2.namedWindow("Mosaic", cv2.WINDOW\_NORMAL)

    cv2.imshow("Mosaic", bgr\_mosaic)

    cv2.waitKey(10)

    # Initialize variables for iterative processing

    bgr\_previous = bgr\_first

    H\_prev\_mosaic = H

    for image\_count in range(1, len(fileNames)):

        bgr\_current = cv2.imread(fileNames[image\_count])

        # Convert images to grayscale

        gray\_previous = cv2.cvtColor(bgr\_previous, cv2.COLOR\_BGR2GRAY)

        gray\_current = cv2.cvtColor(bgr\_current, cv2.COLOR\_BGR2GRAY)

        # Detect and compute features

        kp\_prev, des\_prev = detector.detectAndCompute(gray\_previous, None)

        kp\_curr, des\_curr = detector.detectAndCompute(gray\_current, None)

        # Match descriptors

        matches = matcher.knnMatch(des\_curr, des\_prev, k=2)

        # Apply ratio test to filter good matches

        good\_matches = []

        for m, n in matches:

            if m.distance < 0.8 \* n.distance:

                good\_matches.append(m)

        # Find the homography that maps the current image to the previous image

        src\_pts = np.float32([kp\_curr[m.queryIdx].pt for m in good\_matches]).reshape(-1, 2)

        dst\_pts = np.float32([kp\_prev[m.trainIdx].pt for m in good\_matches]).reshape(-1, 2)

        H\_current\_prev, mask = cv2.findHomography(srcPoints=src\_pts, dstPoints=dst\_pts, method=cv2.RANSAC, ransacReprojThreshold=3.0, maxIters=2000)

        num\_inliers = sum(mask)

        print("Number of inliers: %d" % num\_inliers)

        # Display all matches (optional)

        matchesMask = mask.ravel().tolist()

        bgr\_matches = cv2.drawMatches(img1=gray\_current, keypoints1=kp\_curr, img2=gray\_previous, keypoints2=kp\_prev, matches1to2=good\_matches, matchesMask=matchesMask, outImg=None)

        cv2.imshow("matches", bgr\_matches)

        cv2.waitKey(10)

        # Combine homographies to get the transformation from the current image to the mosaic

        H\_current\_mosaic = H\_prev\_mosaic @ H\_current\_prev

        # Warp the current image to align it with the mosaic

        w = bgr\_mosaic.shape[1]

        h = bgr\_mosaic.shape[0]

        bgr\_current\_warp = cv2.warpPerspective(bgr\_current, H\_current\_mosaic, (w, h))

        # Fuse the current warped image with the mosaic

        bgr\_mosaic = fuse\_color\_images(bgr\_mosaic, bgr\_current\_warp)

        cv2.imshow("Mosaic", bgr\_mosaic)

        cv2.waitKey(0)

        # Update variables for the next iteration

        bgr\_previous = bgr\_current

        H\_prev\_mosaic = H\_current\_mosaic

    # Save the final mosaic image

    cv2.imwrite("mosaic.jpg", bgr\_mosaic)

    print("All done, bye!")

    # Prompt the user to close windows

    while True:

        close = input("Do you want to close all windows? (y/n): ").strip().lower()

        if close == 'y':

            break

    # Properly close all windows and release resources

    cv2.destroyAllWindows()

    cv2.waitKey(1)

def fuse\_color\_images(A, B):

    assert (A.ndim == 3 and B.ndim == 3)

    assert (A.shape == B.shape)

    # Allocate the result image

    C = np.zeros(A.shape, dtype=np.uint8)

    # Create masks for pixels that are not zero

    A\_mask = np.sum(A, axis=2) > 0

    B\_mask = np.sum(B, axis=2) > 0

    # Compute regions of overlap

    A\_only = A\_mask & ~B\_mask

    B\_only = B\_mask & ~A\_mask

    A\_and\_B = A\_mask & B\_mask

    # Fuse the images based on the masks

    C[A\_only] = A[A\_only]

    C[B\_only] = B[B\_only]

    C[A\_and\_B] = 0.5 \* A[A\_and\_B] + 0.5 \* B[A\_and\_B]

    return C

def get\_xy(event, x, y, flags, param):

    if event == cv2.EVENT\_LBUTTONDOWN:

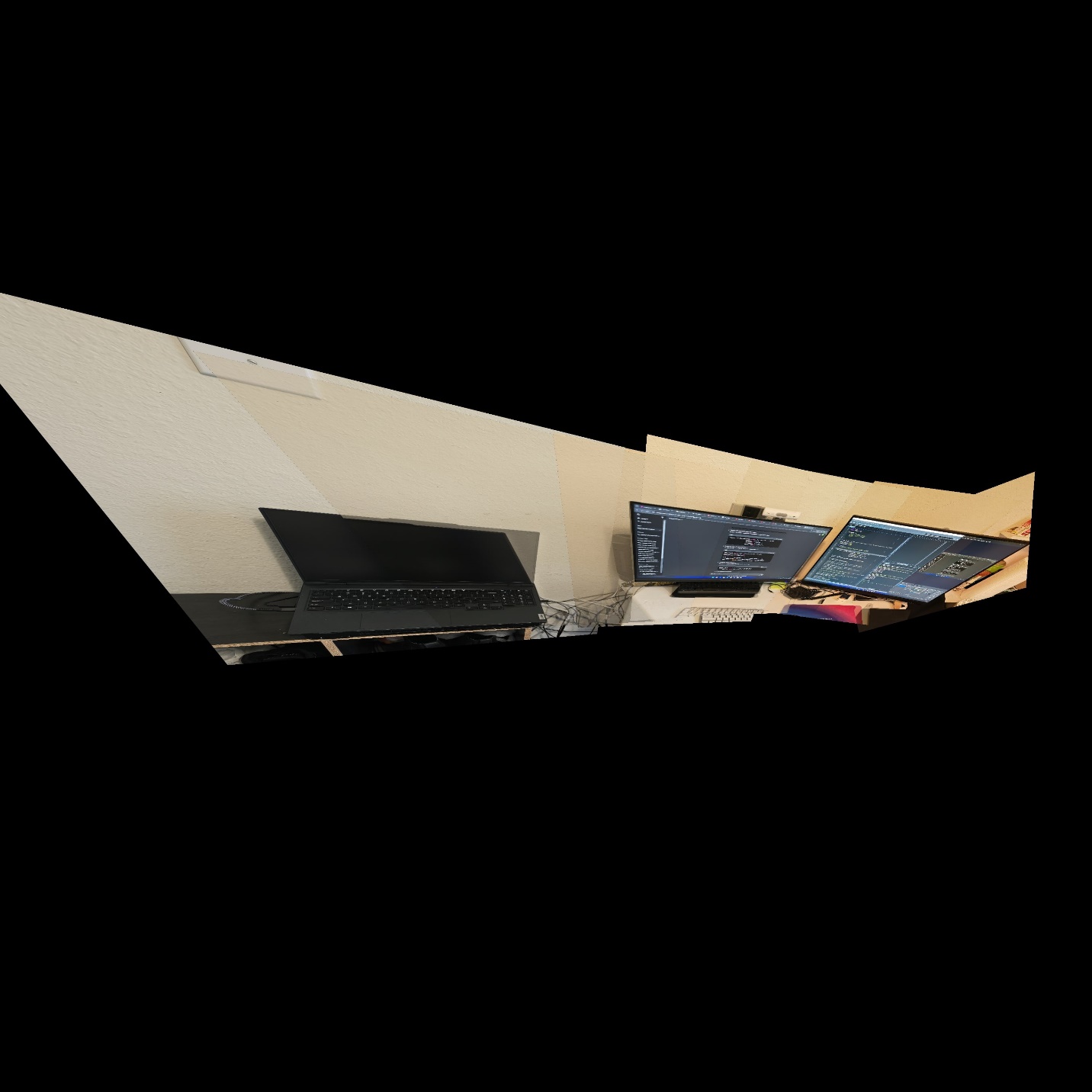
        print(x, y)

        param.append((x, y))

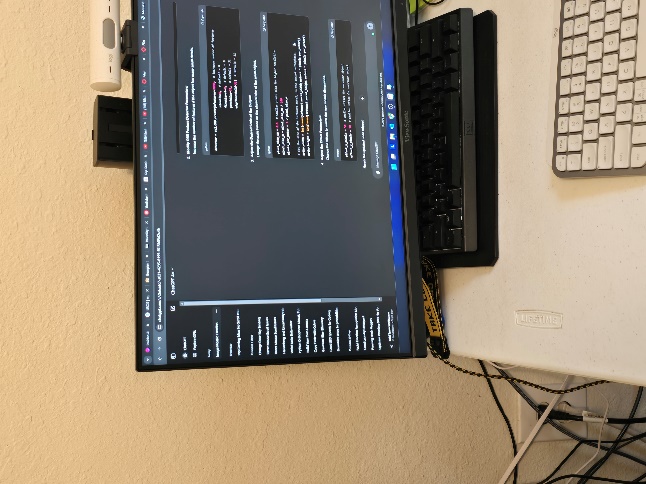
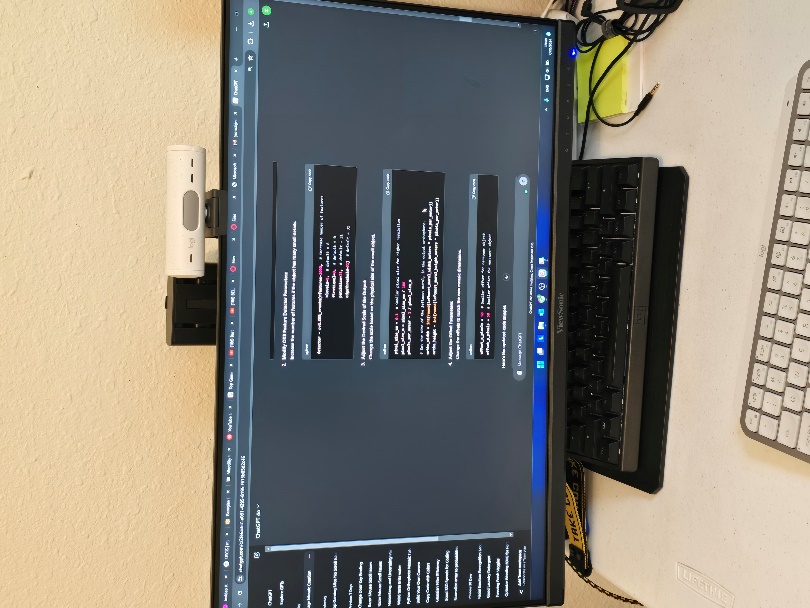
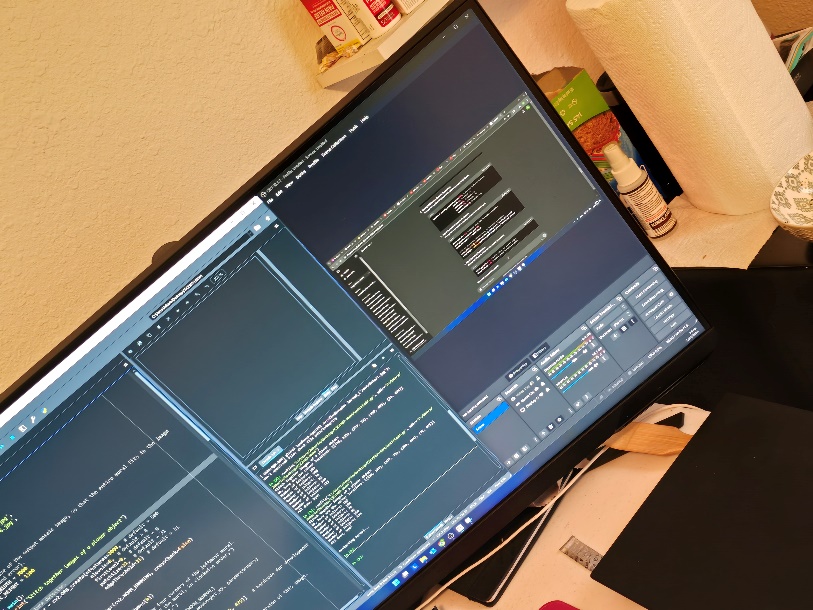
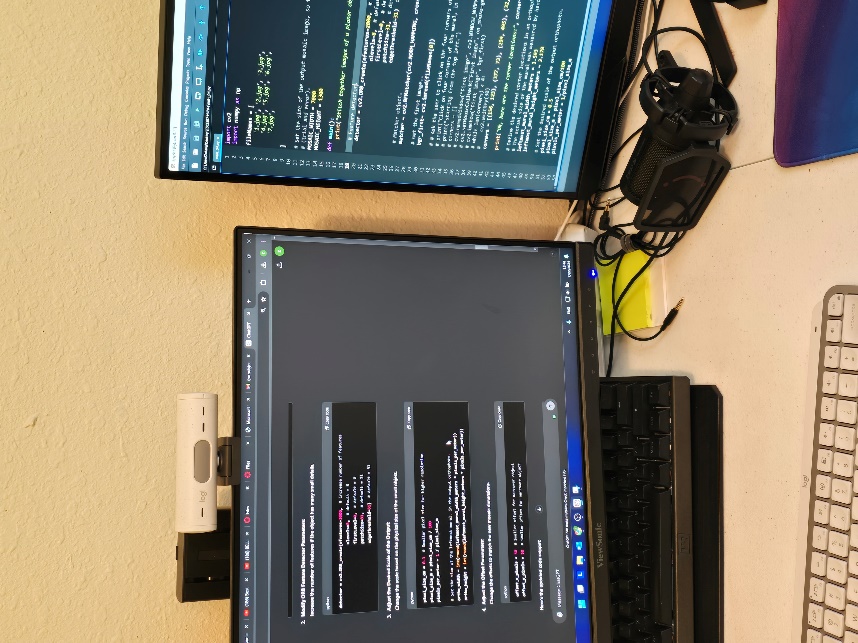
if \_\_name\_\_ == "\_\_main\_\_":

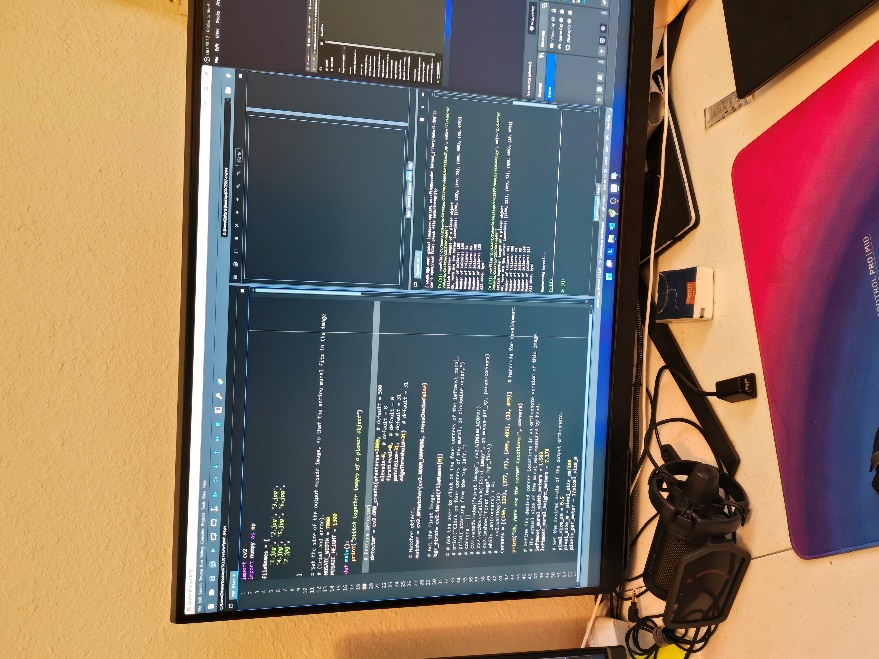
    main()

Second Output:



Second input files:



First input images: