VR Assignment 1

- Nathan Mathew Verghese (IMT2022022)

Tasks

Part - 1: Detect, segment and count coins.

Part - 2: Create a stitched panorama from multiple overlapping images.

Part - 1: Detect, segment and count coins.

The task given was to take an image consisting of coins and detect, segment and count them. My approach was as follows:

1.1 Preprocessing

The image is first converted to grayscale. If this image is directly used for edge detection, then it will detect stray noise. So for this, I used a Gaussian blur so that noise can be reduced well.

Preprocessing and detection

```
images = [cv2.imread('input/' + str(f) +  '.jpg') for f in range(1,4)] # Import images

print(len(images))

images = [cv2.resize(img, (0,0), fx=0.4, fy=0.4) for img in images] # Resize images

g = [cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) for img in images] # Convert to grayscale

g = [cv2.GaussianBlur(img, (9,9), 2) for img in g] # Apply Gaussian blur to reduce noise
```

✓ 0.2s

1.2 Edge Detection and Contour Detection

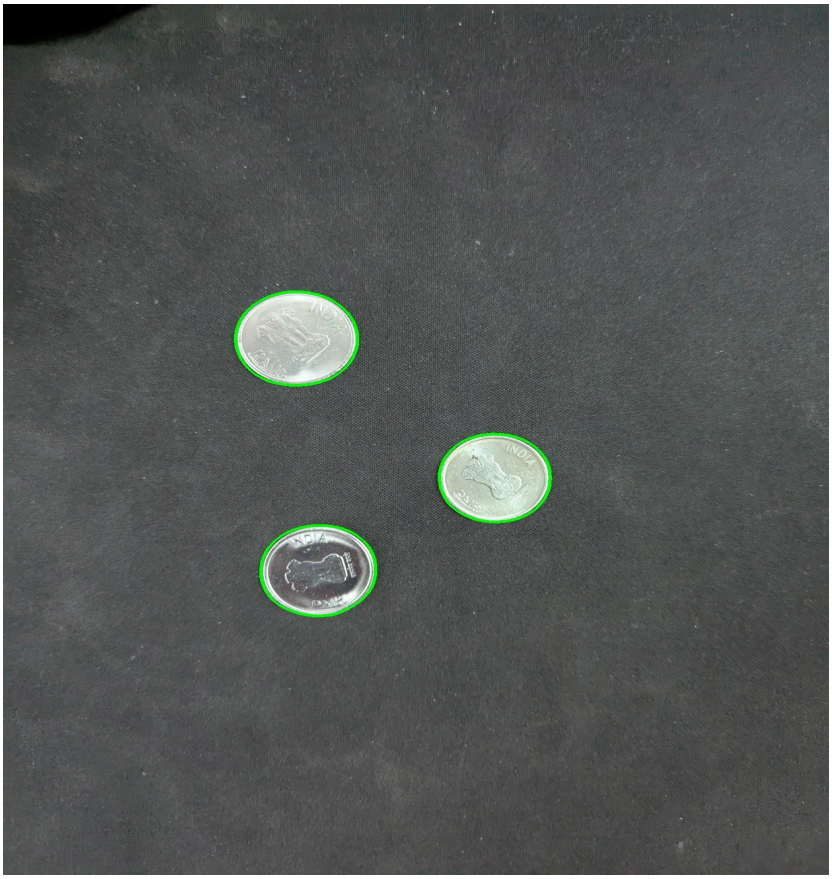
Edges are detected using canny edge detection. Also from that, we detect contours using a kernel and morphological operations. With this, we can get the circular structure of the coin. We then use a threshold to filter circles which are too big or too small.

```
edges = [cv2.Canny(gr, 50, 150) for gr in g] # Use Canny edge detection for each image

kernel = np.ones((3,3), np.uint8) # Create a kernel for morphological operations
dilated = [cv2.dilate(edge, kernel, iterations=2) for edge in edges] # Dilate the edges to make them more visible
closed = [cv2.morphologyEx(dilate, cv2.MORPH_CLOSE, kernel, iterations=2) for dilate in dilated] # Close the edges to make them more solid
```

✓ 0.0s Python

```
contours = [cv2.findContours(close, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE) for close in closed] # Find the contours in the images

# Filter the contours based on their circularity, size and perimeter
detected_contours = [[cnt for cnt in contour[0] if (perimeter := cv2.arcLength(cnt, True)) > 0
                     and 0.7 < (circularity := 4 * np.pi * (cv2.contourArea(cnt) / (perimeter ** 2))) < 1.2
                     and (radius := np.sqrt(cv2.contourArea(cnt) / np.pi)) > 10] for contour in contours]
```

✓ 0.0s Python



Detected edges

1.3 Segmentation and Counting

```python
segmented_coins = []
counter = [] # Counter for the number of coins in each image

j = 0 # Image counter
count = 0 # Coin counter

for img, x in zip(images, detected_contours):
    j += 1 # Increment the image counter
    for i, cnt in enumerate(x):
        (x, y), radius = cv2.minEnclosingCircle(cnt) # Get the minimum enclosing circle of the contour
        center, radius = (int(x), int(y)), int(radius) # Get the center and radius of the circle
        mask = np.zeros_like(img, dtype=np.uint8)
        cv2.circle(mask, center, radius, (255, 255, 255), -1)

        # Ensure the coordinates are within the image boundaries
        y1, y2 = max(center[1] - radius, 0), min(center[1] + radius, img.shape[0])
        x1, x2 = max(center[0] - radius, 0), min(center[0] + radius, img.shape[1])

        coin_segment = cv2.bitwise_and(img, mask)[y1:y2, x1:x2] # Segment the coin from the image
        segmented_coins.append(coin_segment) # Append the segmented coin to the list

        cv2.imwrite(f'output/segmented_{str(j)}_{str(i+1)}.jpg', coin_segment) # Save the segmented coin

    # Display the segmented coins
    plt.figure(figsize=(20, 5))
    plt.title(f'Image {j}')
    plt.axis('off')
    for i, coin in enumerate(segmented_coins[count:]):
        plt.subplot(1, len(segmented_coins[count:]), i+1)
        plt.imshow(cv2.cvtColor(coin, cv2.COLOR_BGR2RGB))
        plt.axis('off')
        # plt.title(f'Image {j} Coin {i+1}')
    plt.show()
    temp = count
    count += len(segmented_coins[count:])
    counter.append(count-temp if count-temp > 0 else count) # Append the number of coins in the image
```

✓ 0.6s

Each detected contour is segmented.

```python
# this has automatically counted the number of coins from the previous part

for i,x in enumerate(counter):
    print(f'Image-{i+1} has {x} coins')

print()
print(f'Total number of coins in all images: {sum(counter)}')
```

✓ 0.0s

Number of coins.

Output

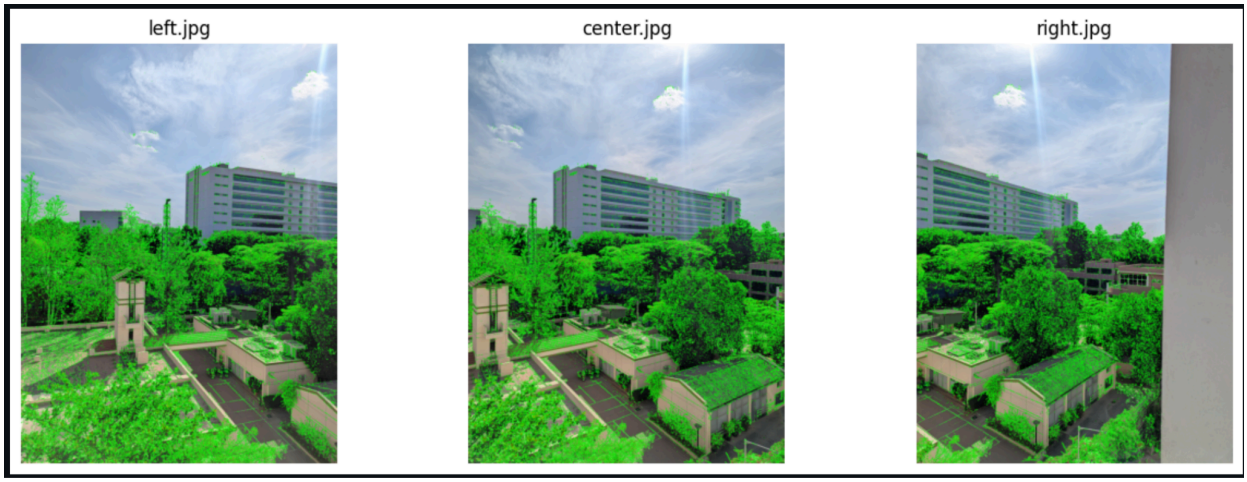
Part - 2: Create a stitched panorama from multiple overlapping images.

The task given was to take multiple images and stitch them together to make a panoramic image.
My approach was as follows:

2.1 Making a SIFT detector and detecting keypoints

I first used cv2 and its detectAndCompute() function to detect keypoints in each image.

```python
def detect_keypoints(image_path):
    image = cv2.imread(image_path) # Load image
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) # Convert to grayscale
    keypoints, descriptors = sift.detectAndCompute(gray, None) # Detect keypoints and descriptors
    return image, keypoints, descriptors
```



Detected keypoints

2.2 Stitching the images using homography

I took each image and stitched them onto the previous one in a sequential manner.

```python
# Perform image stitching
stitched_image = stitch_images(images[0], images[1], keypoints_list[0],keypoints_list[1], descriptors_list[0], descriptors_list[1])

for i in range(1,len(images)):
    gr = cv2.cvtColor(stitched_image, cv2.COLOR_BGR2GRAY)
    kp, dsc = sift.detectAndCompute(gr, None) # Detect keypoints and descriptors
    stitched_image = stitch_images(stitched_image, images[i], kp, keypoints_list[i], dsc, descriptors_list[i]) # Stitch images
    stitched_image = crop_tilted_black_seam(stitched_image) # Crop tilted black seam
```

```python
def stitch_images(image1, image2, keypoints1, keypoints2, descriptors1, descriptors2):
    # Match keypoints
    matcher = cv2.BFMatcher() # Create a matcher object
    matches = matcher.knnMatch(descriptors1, descriptors2, k=2) # Find the 2 nearest neighbors of each descriptor
    good_matches = [m for m, n in matches if m.distance < 0.75 * n.distance] # Here 0.5 is a threshold. this is Lowe's ratio test
    matches = np.array([[keypoints1[m.queryIdx].pt + keypoints2[m.trainIdx].pt] for m in good_matches]).reshape(-1, 4)

    # Compute homography
    points1, points2 = matches[:, :2], matches[:, 2:] # Get points
    H, _ = cv2.findHomography(points2, points1, cv2.RANSAC) # Compute homography

    # Stitch images
    h1, w1 = image1.shape[:2] # Get shape of image1
    h2, w2 = image2.shape[:2]
    panorama = cv2.warpPerspective(image2, H, (w1 + w2, h1)) # Warp image2 to image1
    panorama[0:h1, 0:w1] = image1

    # Crop black areas more precisely
    # gray_panorama = cv2.cvtColor(panorama, cv2.COLOR_BGR2GRAY)
    # _, thresh = cv2.threshold(gray_panorama, 1, 255, cv2.THRESH_BINARY)
    # coords = cv2.findNonZero(thresh)
    # x, y, w, h = cv2.boundingRect(coords)
    # panorama = panorama[y:y+h, x:x+w]

    return panorama
```

In the stitch_images() function, we can see the following happen:

1. Keypoint matching:
    a. A Brute Force Matcher (BFMatcher) is employed to compare descriptors from both images.
    b. For each descriptor in the first image, the two nearest descriptors in the second image are found.
    c. Lowe's ratio test is applied to filter out weak matches, ensuring that only reliable correspondences are retained. This is done using the condition:
        distance of best match < 0.75×distance of second-best match
    d. The matched keypoints are then stored in a structured format, where each match contains corresponding coordinate pairs from both images.
2. Once keypoint correspondences are established, a homography matrix (H) is computed using the RANSAC (Random Sample Consensus) algorithm. This transformation matrix defines how one image should be warped to align with the other.
3. Image Warping and Stitching: With the estimated homography matrix, the second image is warped onto the perspective of the first image. This transformation aligns image2 with image1 in a larger canvas.

2.3 Cropping out black seams

After the stitching function, what was noticed was that there would be black seams in the image because the second image would be warped to the first's perspective. So to fix the issue of these seams showing up in the final image, I cropped out the seam. This will lose some information about the image (a minor amount) but at least it will retain the integrity and visibility of the image. The code for this is attached below.

```python
def crop_tilted_black_seam(image):
    gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY) # Convert to grayscale

    _, thresh = cv2.threshold(gray, 1, 255, cv2.THRESH_BINARY) # Threshold to detect black pixels

    contours, _ = cv2.findContours(255 - thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE) # Find contours of the black region

    if not contours:
        return image  # No black regions found, return original image

    largest_contour = max(contours, key=cv2.contourArea) # Find the largest black region (the seam)

    rect = cv2.minAreaRect(largest_contour) # Get bounding rotated rectangle
    box = cv2.boxPoints(rect) # Get box points
    box = np.int0(box) # Convert to integers

    x, y, w, h = cv2.boundingRect(largest_contour) # Get the bounding box
    cropped = image[:, :x]  # Crop everything before the seam starts

    return cropped
```

2.4 Final output

The final output is attached above.

Observations

Part - 1:

In part 1, a good amount of preprocessing was needed before the edge detection could be done. I found the best parameters through hit and trial.

Part - 2:

In part 1, I couldn't directly stitch the images due to the warp issue. Alternatively, the stitcher functions in the cv2 library could have been used, but this would then sort of defy the principle of learning in the assignment, so I decided against using this.

Code

Refer to https://github.com/nathanmathewv/VR_Assignment1_NathanMathew_IMT2022022