

2024 COMP6528

Computer Vision

Assignment 3 Report

Name: Xing Chen

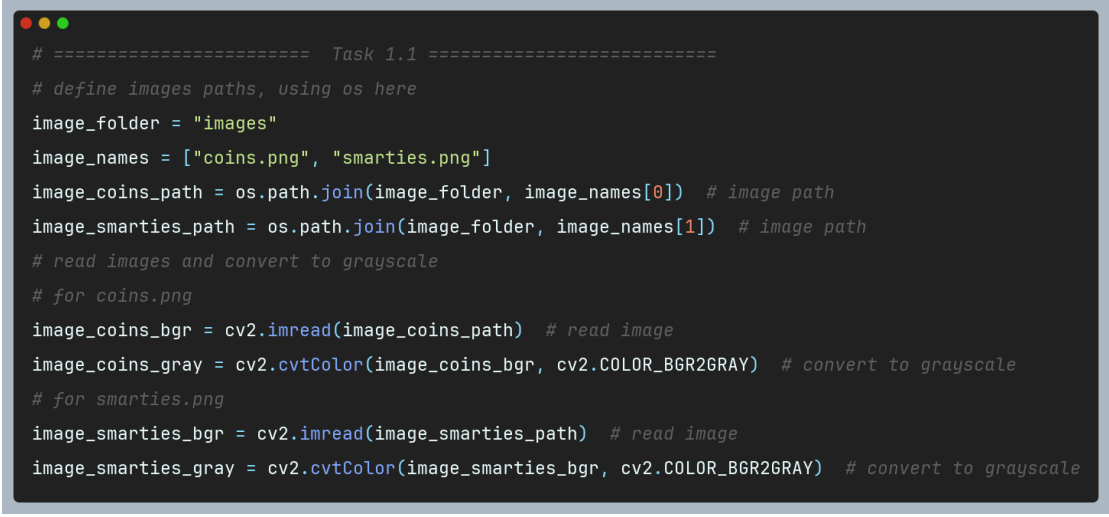
UID: u7725171

1 Task1 Model Fitting with the Hough Circle Transform

To improve code readability and encapsulation, the implementation for Task 1 was separated into two parts: the core Hough transform and non-maximum suppression functions were placed in `HoughCircleTransform.py`, while the main execution flow was in `Task1.py`.

1.1 Read images and convert to grayscale.

As shown in Figure 1-1, the RGB images were read in as NumPy arrays using the `cv2.imread()` function and then were converted Grayscale.



```
# ===== Task 1.1 =====  
# define images paths, using os here  
image_folder = "images"  
image_names = ["coins.png", "smarties.png"]  
image_coins_path = os.path.join(image_folder, image_names[0]) # image path  
image_smarties_path = os.path.join(image_folder, image_names[1]) # image path  
# read images and convert to grayscale  
# for coins.png  
image_coins_bgr = cv2.imread(image_coins_path) # read image  
image_coins_gray = cv2.cvtColor(image_coins_bgr, cv2.COLOR_BGR2GRAY) # convert to grayscale  
# for smarties.png  
image_smarties_bgr = cv2.imread(image_smarties_path) # read image  
image_smarties_gray = cv2.cvtColor(image_smarties_bgr, cv2.COLOR_BGR2GRAY) # convert to grayscale
```

Figure 1-1 images read and conversion.

1.2 Apply Canny edge detection

It's very convenient to use canny edge detection here.

The Canny edge detection treats those pixels with gradient higher than high threshold as edges. Pixels with gradient lower than low threshold will be ignored. Only pixels those are neighboring with edge pixels and have gradient between two thresholds will be recon as edges.

To find the best parameters, I plot the gradient histogram of the images as shown in Figure 1-1 and Figure 1-2 to find the proper low and high threshold.

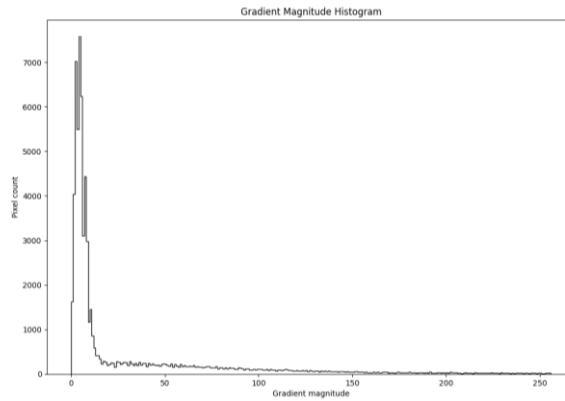


Figure 1-2 gradient histogram of coins.png

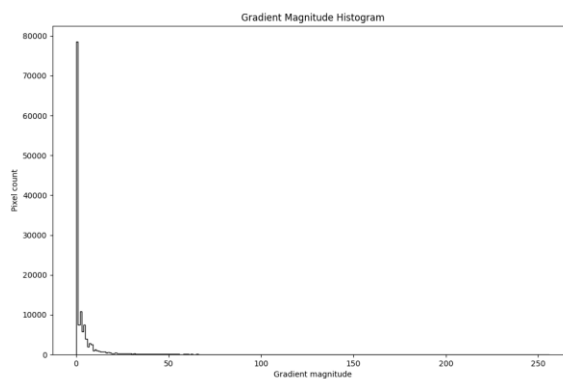


Figure 1-3 gradient histogram of smarties.png

According to the histograms, a proper threshold for coins.png is around between (100, 200), while for smarties the best choice is about (25,60). After fine-tuning, I choose to use (108, 211) and (30, 65) for coins.png and smarties.png respectively.

Below in Figure 1-4 is the code used to apply canny edge detection.

```
# ===== Task 1.2 =====
# apply canny edge detection on images
image_coins_edges = cv2.Canny(image_coins_gray, 108, 211) # use canny edge detect of cv2
image_smarties_edges = cv2.Canny(image_smarties_gray, 30, 65) # use canny edge detect of cv2
```

Figure 1-4 canny edge detection.

1.3 Hough Transform Function

To implement this function, use the algorithm below:

```
for each pixel in edge image:
    for each possible r:
```

```

        iterate in 360 degrees:
            calculate a, b
            accumulate the voting if in image range
        end
    end
end

```

The code of this function is shown below in Figure 1-5.

```

def hough_circle_transform(edges, min_radius, max_radius, radius_step, angle_step, threshold_ratio):
    """
    Task1.3
    Apply hough circle transformation on an edge image.
    """
    # Get the dimensions of the input image
    height, width = edges.shape

    # Initialize the accumulator array
    num_radii = (max_radius - min_radius) // radius_step + 1 # Calculate number of radius
    accumulator = np.zeros(shape=(height, width, num_radii), dtype=np.uint64)

    # Get the indices of edge pixels
    edge_pixels = np.argwhere(edges != 0)

    # precompute angles and cos, sin
    angles = np.arange(0, 360, angle_step)
    cos_theta = np.cos(np.deg2rad(angles))
    sin_theta = np.sin(np.deg2rad(angles))

    # Iterate over all edge pixels and radius
    for x, y in tqdm(edge_pixels, desc="Processing edge pixels", total=len(edge_pixels)):
        for radius_index, radius in enumerate(range(min_radius, max_radius, radius_step)):
            for cos_t, sin_t in zip(cos_theta, sin_theta):
                a = int(x - radius * cos_t) # Calculating a using formula a = x - r cos θ
                b = int(y - radius * sin_t) # Calculating b using formula b = y - r sin θ
                # If within this image range, add the score by 1
                if 0 ≤ a < height and 0 ≤ b < width:
                    accumulator[a, b, radius_index] += 1

    # Find the maxima in the accumulator using threshold ratio
    max_acc = np.max(accumulator)
    threshold = threshold_ratio * max_acc
    circles = [] # List to save the result

    # Apply thresholding
    for r in range(accumulator.shape[2]):
        acc_slice = accumulator[:, :, r] # get a slice of the accumulator
        circle_indices = np.argwhere(acc_slice ≥ threshold) # get the indices of circles with score is over threshold
        for idx in circle_indices:
            a, b = idx
            circles.append([a, b, min_radius + r * radius_step]) # append the circle into result List

    # Sort by descending order of the vote score of each circle
    circles = sorted(circles, key=lambda x: accumulator[x[0], x[1], (x[2] - min_radius) // radius_step], reverse=True)

    return circles

```

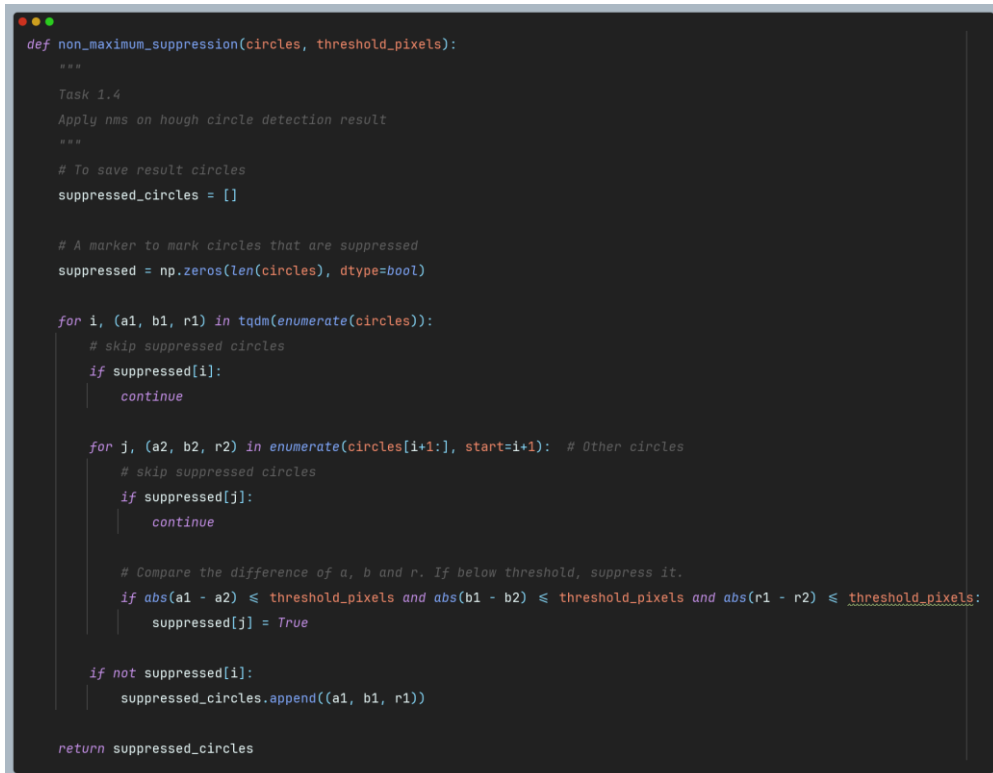
Figure 1-5 code of hough circle transform.

1.4 NMS

For the non-maximum suppression, the algorithm below was implemented:

```
for each circle in the circles:
    compare the circles with the remaining circles:
        check the difference between them
        if difference threshold:
            suppress the latter
    end
end
```

Code screenshot is listed below in Figure 1-7.



```
def non_maximum_suppression(circles, threshold_pixels):
    """
    Task 1.4
    Apply nms on hough circle detection result
    """
    # To save result circles
    suppressed_circles = []

    # A marker to mark circles that are suppressed
    suppressed = np.zeros(len(circles), dtype=bool)

    for i, (a1, b1, r1) in tqdm(enumerate(circles)):
        # skip suppressed circles
        if suppressed[i]:
            continue

        for j, (a2, b2, r2) in enumerate(circles[i+1:], start=i+1): # Other circles
            # skip suppressed circles
            if suppressed[j]:
                continue

            # Compare the difference of a, b and r. If below threshold, suppress it.
            if abs(a1 - a2) <= threshold_pixels and abs(b1 - b2) <= threshold_pixels and abs(r1 - r2) <= threshold_pixels:
                suppressed[j] = True

        if not suppressed[i]:
            suppressed_circles.append((a1, b1, r1))

    return suppressed_circles
```

Figure 1-6 NMS code.

1.5 Result from 1.1-1.4

After implementing all the code in previous tasks, parameters below in Table 1-1 were used and then plot the raw result here.

Image	Radius range	Radius step	Theta step	Threshold ratio	NMS threshold
coins	(10,100)	1	2	0.75	10

smarties	(10,100)	1	2	0.75	10
----------	----------	---	---	------	----

Table 1-1 parameters before adjustment.

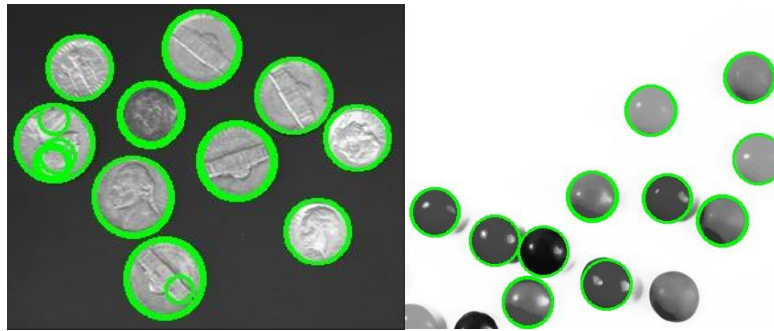


Figure 1-7 circle detected with Hough circle transform.

From Figure 1-8 above, circle transform had detected almost all of the circles in images, but with multiple circles within or around the target.

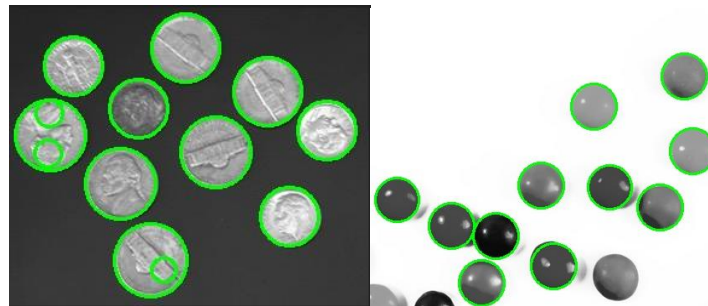


Figure 1-8 circle detected with NMS

Using NMS, multiple circles are eliminated, but there are still some circles we don't want. (e.g. Figure 1-9)

1.6 Assessment on result

1.6.1 Raw result assessment

For coins.png, we can see that there are 10 coins in it. All 10 coins are detected, but 3 false circles are detected too.

- **False negative rate:** 0%.
- **False positive rate:** 23.1%.
- **Accuracy:** 76.9%.

For smarties.png, we can see that there are 14 smarties in it. 11 smarties are detected, missing 3 smarties. No false circles.

- **False negative rate:** 21.4%.

- **False positive rate:** 0%.
- **Accuracy:** 100%.

Effectiveness and shortcomings of the result.

- **Effectiveness:** The algorithm performs well on recognizing desired items.
- **Shortcomings:**
 - Image with complex texture will cause false positive. This need to be improved.
 - Some items cannot be detected.
 - Single set of parameters is not universal.

1.6.2 Adjustment on parameters

To adjust the result, we need to modify the parameters.

Finally, fetching the accuracy and preserving the time efficiency at the same time, I choose the parameters as in Table 1-2 below to better fit the task:

- Higher the min_radius to 25 to ignore circles that are smaller than our item.
- Lower the max_radius to 50 to increase time efficiency.
- Increase the radius step to 2 and angle step to 10 to increase time efficiency.
- Lower threshold ratio to 0.7 and 0.6 respectively to include items that are not detected but should be.
- The NMS threshold distance was set to 10 pixels, as this value eliminated duplicate detections while preserving adjacent distinct circles.

Image	Radius range	Radius step	Theta step	Threshold ratio	NMS threshold
coins	(25,50)	2	10	0.7	10
smarties	(25,50)	2	10	0.6	10

Table 1-2 parameters after adjustment.

1.6.3 Final result

Using this parameters, we can see that the result is much better in Figure 1-9 and 1-10, with 100% accuracy upon coins and very high accuracy upon smarties.

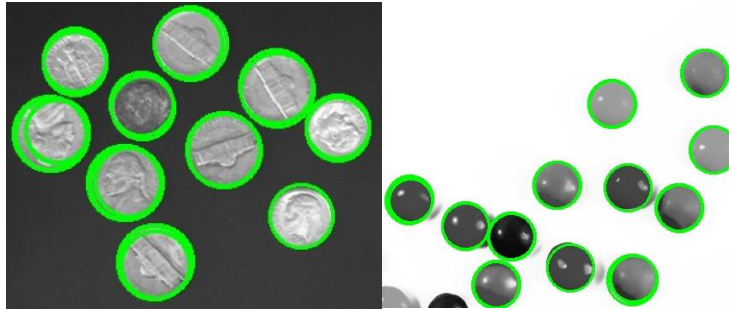


Figure 1-9 Hough circle detection after parameter optimization.

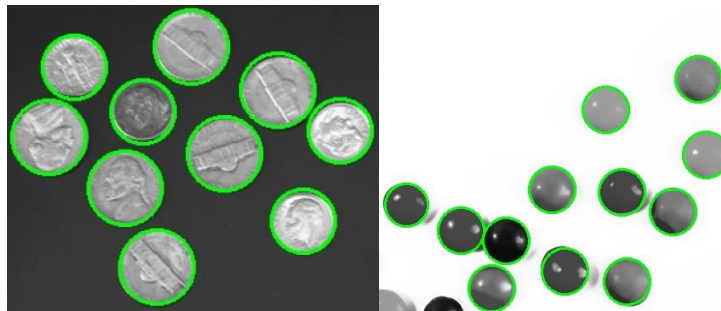


Figure 1-10 NMS on circle detection after parameter optimization.

Almost all circle are detected and no false positive are produced. The time efficiency has also increased dramatically as we can see in Figure 1-11.

```
Hough circle transformation: processing coins.png
Processing edge pixels: 100%|██████████| 5162/5162 [00:03<00:00, 1460.58it/s]
Processing edge pixels: 0%|          | 0/4117 [00:00<?, ?it/s]Hough circle transformation: processing smarties.png
Processing edge pixels: 100%|██████████| 4117/4117 [00:02<00:00, 1499.55it/s]
```

Figure 1-11 only 3 seconds are needed.

1.6.4 Circles not in image

I also noticed that some circle that are not totally in the smarties image are ignored. If we want to keep them, we need to adjust our algorithm like shown in Figure, Figure and Figure.

```
# Initialize the accumulator array
max_offset = max_radius #
acc_height = height + 2 * max_offset
acc_width = width + 2 * max_offset
num_radii = (max_radius - min_radius) // radius_step + 1 # Calculate number of radius
accumulator = np.zeros(shape=(acc_height, acc_width, num_radii), dtype=np.uint64)
```

Figure 1-12 Expanding the accumulator by an offset.


```

# Apply thresholding
for r in range(accumulator.shape[2]):
    acc_slice = accumulator[:, :, r] # get a slice of the accumulator
    circle_indices = np.argwhere(acc_slice >= threshold) # get the indices of circles with score is over threshold
    for idx in circle_indices:
        a, b = idx
        a -= max_offset # adjust the offset to go back to original coordinates
        b -= max_offset # adjust the offset to go back to original coordinates
        circles.append([a, b, min_radius + r * radius_step]) # append the circle into result List

```

Figure 1-13 set a, b back to original coordinates.

1.7 Impact of different parameters

This part we will discuss the impact of different parameters of the Hough circle transform. We will supplement them with pictures. All parameters in this section are the same as Table 1-2 in 1.6.2.

- **Radius range:** Table 1-3
 - **Time Efficiency:** Expanding the range will increase time cost of the algorithm. **Impact level: high.**
 - **Result Accuracy:** If the range is not proper (e.g. min range higher than the items) , will loose information. **Impact level: high.**

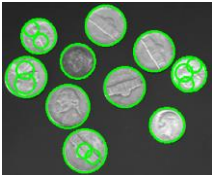
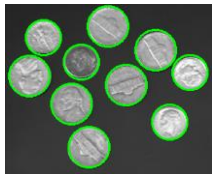
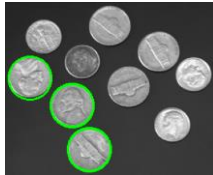
image	Radius range	(10, 100)	(25, 50)	(30, 40)
coins	Accuracy			
	Run time	11s	4s	1s

Table 1-3

- **Radius step:** Table 1-4
 - **Time Efficiency:** Higher step will dramatically reduce time cost. **Impact level: high.**
 - **Result Accuracy:** Expanding the radius step has significantly impact on item detecting (e.g. losing circles or circle with offset). Better keep it 1. **Impact level: high.1**

image	Radius step	1	2	5
-------	-------------	---	---	---

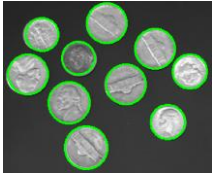
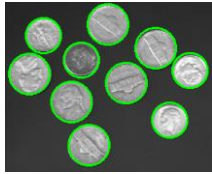
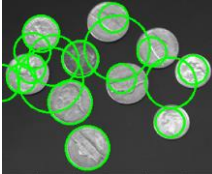
coins	Accuracy			
	Run time	8s	4s	<1s

Table 1-4

- **Theta step:** Table 1-5
 - **Time Efficiency:** Higher step will dramatically reduce time cost.
Impact level: high.
 - **Result Accuracy:** Expanding the theta step in a proper range has less impact on result. **Impact level: medium.**

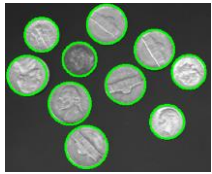
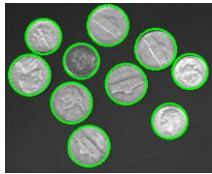
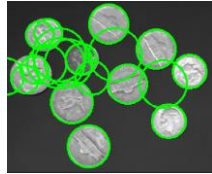
image	Theta step	5	10	20
coins	Accuracy			
	Run time	8s	4s	2s

Table 1-5

- **Threshold ratio:** Table 1-6
 - **Time Efficiency:** **Impact Level: low.**
 - **Result Accuracy:** Low threshold will include noise in to result and high threshold will lose vital information. **Impact level: high.**

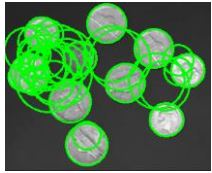
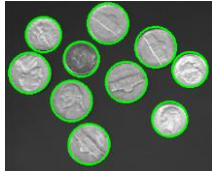
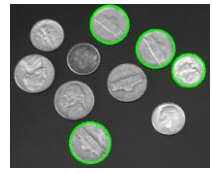
image	Threshold ratio	0.5	0.7	0.9
coins	Accuracy			
	Run time	4s	4s	3s

Table 1-6

- **NMS threshold:** Table 1-7
 - **Time Efficiency:** **Impact Level: ignorable.**
 - **Result Accuracy:** Huge numbers will cause circles that are close to or overlapping with each other to be suppressed. On the other hand, small values of threshold can't detect some of the circles that should be suppressed. **Impact level: high.**

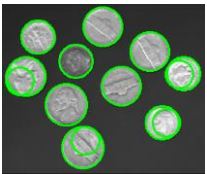
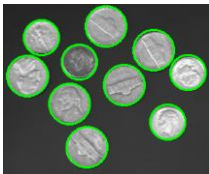
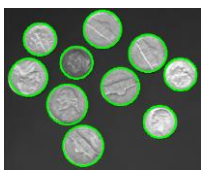
image	NMS threshold	5	10	20
coins	Accuracy			

Table 1-7

1.8 Time and memory efficiency

1.8.1 hough_circle_transform

Time complexity - $O(E * R * (360 / \text{angle_step}))$

- initialization:
 - initializing accumulator - $O(H * W * R)$
 - calculating cos and sin - $O(360 / \text{angle_step})$
- iterate edge pixels: $O(E)$
- iterate radius: $O(R)$
- iterate angle: $O(360 / \text{angle_step})$

where E represents the number of edge pixels, R is the number of radius.

Space complexity - $O(H * W * R)$

- accumulator: $O(H * W * R)$
- rad array: $O(360 / \text{angle_step})$

1.8.2 non_maximum_suppression

Time complexity - $O(C^2)$

- iterate circles: $O(C)$
- compare with other circles: $O(C)$

where C is the numbers of circles.

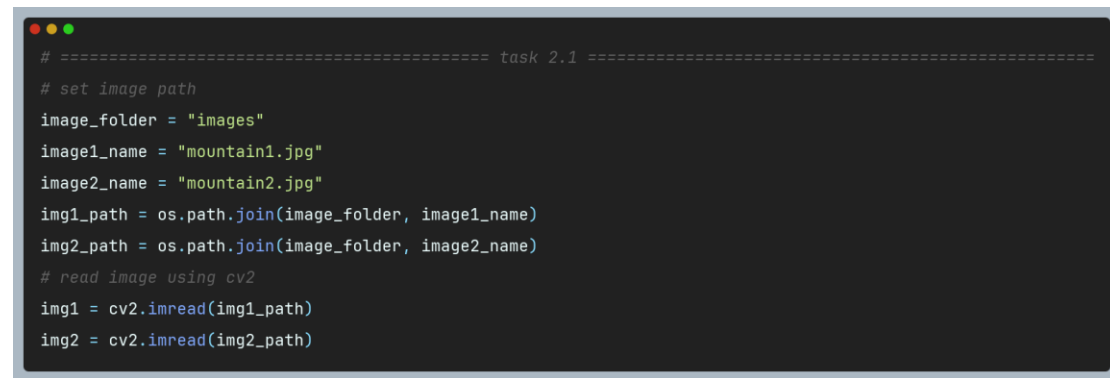
Space complexity - $O(C)$

- marker 'supressed' array: $O(C)$
- output 'suppressed_circles' array: $O(C)$

2 Task2 Two-view Homography Estimation

2.1 Read images

As shown in the code in Figure 2-1, two images were read using cv2.



```
# ===== task 2.1 =====  
# set image path  
image_folder = "images"  
image1_name = "mountain1.jpg"  
image2_name = "mountain2.jpg"  
img1_path = os.path.join(image_folder, image1_name)  
img2_path = os.path.join(image_folder, image2_name)  
# read image using cv2  
img1 = cv2.imread(img1_path)  
img2 = cv2.imread(img2_path)
```

Figure 2-1 images reading using cv2.

2.2 Homography function

My function implements the algorithm below:

```
For each correspondence coordinates in image 1 and 2:  
    Compute matrix A  
Assemble A  
Compute SVD of A  
Take the last column of V and reshape it as H  
Normalize H
```

The code is listed below in Figure 2-2.

```

def homography(u1, v1, u2, v2):
    """
        Task 2.2

        Computes the homography H using the Direct Linear Transformation
        Arguments:
        u1, v1: normalised (u,v) coordinates from image 1
        u2, v2: normalised (u,v) coordinates from image 2
        Output:
        H: the 3x3 homography matrix that warps normalised coordinates
        from image 1 into normalised coordinates from image 2
    """

    # Assert that the input coordinates have the same length
    assert len(u1) == len(v1) == len(u2) == len(v2), "Input coordinates must have the same length"
    # get the number of points, at least 4
    num_points = len(u1)
    if num_points < 4:
        raise ValueError("At least 4 points are needed")
    A = [] # initialize A
    for i in range(num_points): # in each pair of points
        x1, y1 = u1[i], v1[i]
        x2, y2 = u2[i], v2[i]

        # Construct the coefficient matrix A of the linear system
        A.append([-x1, -y1, -1, 0, 0, 0, x1 * x2, y1 * x2, x2])
        A.append([0, 0, 0, -x1, -y1, -1, x1 * y2, y1 * y2, y2])
    # Convert A to a numpy array
    A = np.array(A)
    # Perform SVD decomposition on A
    U, S, Vh = np.linalg.svd(A)
    L = Vh[-1, :] / Vh[-1, -1] # The homography matrix H is the last row of Vh
    H = L.reshape(3, 3) # reshape to 3x3
    return H

```

Figure 2-2homography function.

2.3 Transform unnormalized points.

The algorithms used in these two functions are:

```

def compute_normalisation_matrix:
    Compute the centroid of these points
    Shift the points to center the centroid at the origin
    Compute average distance and then the scale factor
    Construct the normalization matrix T

def homography_w_normalisation:
    compute the normalization matrixes of two points set
    normalize the coordinates using the matrixes
    compute homography H
    denormalized H using  $H = T2^{(-1)} * H_{norm} * T1$ 

```

Code used in this section is listed below in Figure 2-3 and 2-4.

```
def compute_normalisation_matrix(points):  
    """  
        Task 2.3  
  
        Computes the normalization transformation matrix for homogeneous point coordinates.  
        Arguments:  
        points: Array of points with shape (N, 2) where N is the number of points.  
        Output:  
        T: The normalization transformation matrix.  
    """  
  
    mean = np.mean(points, axis=0) # Compute the mean of the points  
    std_dev = np.std(points, axis=0) # Compute the standard deviation of the points  
    # Compute the normalization scale factor  
    scale = np.sqrt(2) / std_dev  
    # Construct the normalization transformation matrix T  
    T = np.array([[scale[0], 0, -scale[0] * mean[0]],  
                  [0, scale[1], -scale[1] * mean[1]],  
                  [0, 0, 1]])  
  
    return T
```

Figure 2-3 compute normalization matrix

```
def homography_w_normalisation(u1, v1, u2, v2):  
    """  
        Task 2.3  
  
        Computes the homography matrix with normalization using the DLT algorithm.  
        Arguments:  
        u1, v1: normalised (u,v) coordinates from image 1  
        u2, v2: normalised (u,v) coordinates from image 2  
        Output:  
        H: the 3x3 homography matrix that warps normalised coordinates from image 1 into normalised coordinates from image 2  
    """  
  
    # Stack the input coordinates into point arrays  
    points1 = np.stack( arrays=(u1, v1), axis=-1)  
    points2 = np.stack( arrays=(u2, v2), axis=-1)  
    # Compute the normalization transformation matrices T1 and T2  
    T1 = compute_normalisation_matrix(points1)  
    T2 = compute_normalisation_matrix(points2)  
    # Normalize the point coordinates  
    ones = np.ones((points1.shape[0], 1))  
    normalized_points1 = (T1 @ np.hstack((points1, ones))).T.T  
    normalized_points2 = (T2 @ np.hstack((points2, ones))).T.T  
    # Extract the normalized point coordinates  
    u1_norm = normalized_points1[:, 0]  
    v1_norm = normalized_points1[:, 1]  
    u2_norm = normalized_points2[:, 0]  
    v2_norm = normalized_points2[:, 1]  
    # Compute the homography matrix H_norm using the normalized coordinates  
    H_norm = homography(u1_norm, v1_norm, u2_norm, v2_norm)  
    # De-normalize the homography matrix  
    H = np.linalg.inv(T2) @ H_norm @ T1  
    H = H / H[2, 2] # Normalize the homography matrix so that its last element is 1  
  
    return H
```

Figure 2-4 homography_w_normalisation

2.4 SIFT

To extract features, cv2.SIFT was used to detect and compute the SIFT features.

To find matching feature points across the images, BFMatcher was found useful.

To get the sample points, seed 42 was used to randomly sample half of the matched points. The code is listed below in Figure 2-5.

```
def match_and_save(img1, img2, save_path1, save_path2):
    # Task 2.4
    # Detects SIFT keypoints, performs feature matching, and saves the coordinates of matched points.
    # Convert the images to grayscale
    gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
    # Initialize the SIFT feature detector
    sift = cv2.SIFT_create()
    # Detect SIFT keypoints and compute descriptors
    keypoints_1, descriptors_1 = sift.detectAndCompute(img1, None)
    keypoints_2, descriptors_2 = sift.detectAndCompute(img2, None)
    # Use BFMatcher for feature matching
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    matches = bf.match(descriptors_1, descriptors_2)
    # Sort the matches based on feature distance
    matches = sorted(matches, key=lambda x: x.distance)
    np.random.seed(42) # Set a fixed random seed for reproducibility
    # Randomly select half of the matches
    num_matches = len(matches)
    selected_matches = np.random.choice(matches, size=num_matches // 2, replace=False)
    # Extract the coordinates of the matched points
    points1 = np.zeros((len(selected_matches), 2))
    points2 = np.zeros((len(selected_matches), 2))
    for i, match in enumerate(selected_matches):
        points1[i, :] = keypoints_1[match.queryIdx].pt
        points2[i, :] = keypoints_2[match.trainIdx].pt
    # Save the coordinates of the matched points as .npy files
    np.save(save_path1, points1)
    np.save(save_path2, points2)
```

Figure 2-5 get sample points.

2.5 The RANSAC algorithm

The RANSAC algorithm is like this:

Randomly select 4 point pairs from the matched feature points between image 1 and image 2.

Compute the homography matrix H using the selected 4 point pairs and the normalized DLT algorithm.

```
Apply the computed homography matrix  $H$  to transform all the points from
image 1.
Compute the Euclidean distance (error) between the transformed points from
image 1 and their corresponding points in image 2.
Classify the point pairs as inliers if their distance (error) is below a
specified threshold.
Update the best homography matrix if the current iteration yields a higher
number of inliers than the previous best.
Repeat steps 1-6 for a maximum number of iterations or until the maximum
number of iterations is reached.
Return the best homography matrix found across all iterations.
```

The code is listed in Figure 2-6.


```

def homography_w_normalisation_ransac(u1, v1, u2, v2, threshold=1.0, max_iterations=1000):
    """
        Task 2.5

        Computes the homography matrix using the RANSAC algorithm and normalized DLT algorithm
        Arguments:
        u1, v1: (u,v) coordinates from image 1
        u2, v2: (u,v) coordinates from image 2
        threshold: Distance threshold for the RANSAC algorithm, default is 1.0
        max_iterations: Maximum number of iterations for the RANSAC algorithm, default is 1000
        Output:
        best_H: The best homography matrix computed using the RANSAC algorithm and normalized DLT algorithm
    """

    num_points = len(u1)
    best_H = None
    max_inliers = 0

    for _ in range(max_iterations):
        # Randomly select 4 points
        indices = np.random.choice(num_points, size=4, replace=False)
        u1_sample = u1[indices]
        v1_sample = v1[indices]
        u2_sample = u2[indices]
        v2_sample = v2[indices]

        # Compute the homography matrix H using the selected 4 points
        H = homography_w_normalisation(u1_sample, v1_sample, u2_sample, v2_sample)
        # Stack the point coordinates into arrays
        ones = np.ones((num_points, 1))
        points1 = np.stack(arrays=(u1, v1), axis=-1)
        points2 = np.stack(arrays=(u2, v2), axis=-1)
        # Apply the homography matrix H to transform the points
        projected_points = H @ np.vstack((points1.T, np.ones(num_points)))
        projected_points /= projected_points[2, :]
        projected_points = projected_points[:2, :].T
        # Compute the errors between the projected points and the actual points
        errors = np.sqrt(np.sum((points2 - projected_points) ** 2, axis=1))
        # Determine the inliers (points with errors below the threshold)
        inliers = errors < threshold
        num_inliers = np.sum(inliers)
        # Update the best homography matrix and the maximum number of inliers
        if num_inliers > max_inliers:
            max_inliers = num_inliers
            best_H = H

    return best_H

```

Figure 2-6 homography_w_normalisation_ransac.

2.6 Warp and stitch

2.6.1 algorithm

To implement the warp and stitch function, I followed these steps:

Use the homography matrix to transform the corner points of the first image.

Calculate the minimum and maximum coordinates of the transformed corners of the first image and the corners of the second image.

Use these coordinates to determine the size of the resulting stitched image.

Use the homography matrix to warp the first image to align with the second image.

Overlay the warped first image onto the second image to create the stitched image.

The code for the warp and stitch function is shown below in Figure 2-7:

```
def warp_and_stitch(img1, img2, H):  
    """  
        Task 2.6  
        Warps and stitches the images using the homography matrix  
        Arguments:  
        img1: Image 1  
        img2: Image 2  
        H: Homography matrix  
        Output:  
        result_img: The resulting image after warping and stitching  
    """  
    # Get the height and width of the images  
    h1, w1 = img1.shape[:2]  
    h2, w2 = img2.shape[:2]  
    # Define the four corner coordinates of image 1  
    corners_img1 = np.array([ [0, 0], [w1, 0], [w1, h1], [0, h1]], dtype='float32')  
    # Transform the corner points of image 1 using the homography matrix H  
    corners_transformed_img1 = cv2.perspectiveTransform(np.array([corners_img1]), H)[0]  
    # Define the four corner coordinates of image 2  
    corners_img2 = np.array([ [0, 0], [w2, 0], [w2, h2], [0, h2]], dtype='float32')  
    # Combine the transformed corner points of image 1 and the corner points of image 2  
    all_corners = np.vstack((corners_transformed_img1, corners_img2))  
    # Compute the minimum and maximum coordinate values of the combined corner points  
    [xmin, ymin] = np.int32(all_corners.min(axis=0) - 0.5)  
    [xmax, ymax] = np.int32(all_corners.max(axis=0) + 0.5)  
    # Compute the translation amounts  
    t = [-xmin, -ymin]  
    H_translation = np.array([[1, 0, t[0]], [0, 1, t[1]], [0, 0, 1]]) # Construct the translation matrix  
    # Apply perspective transformation and translation to image 1  
    result_img = cv2.warpPerspective(img1, H_translation @ H, dsize=(xmax - xmin, ymax - ymin))  
    # Copy image 2 to the corresponding position in the resulting image  
    result_img[t[1]:h2 + t[1], t[0]:w2 + t[0]] = img2  
    return result_img
```

Figure 2-7 function for warp and stitch

2.6.2 Result

After processing and stitching the images mountain1.jpg and mountain2.jpg, the result is shown in Figure 2-8. The homography estimation successfully aligned and blended the overlapping areas of the two images to produce a seamless stitched image. The visual inspection confirms that the stitching process effectively merged the images,

preserving the continuity of the scene.



Figure 2-8 Result

2.7 Discuss the factors that affect the rectified results theoretically and empirically.

2.7.1 Theoretical Analysis

Quality of Feature Extraction and Matching

- **Number and Distribution of Feature Points:** The quantity and distribution of feature points impact the quality of matching. If there are too few feature points or if they are unevenly distributed, it may be difficult to find enough matching points, leading to registration failures or inaccuracies.
- **Discriminability and Robustness of Feature Descriptors:** The distinctiveness and robustness of feature descriptors are also crucial. Good feature descriptors should remain stable under image transformations (such as rotation and scaling) and noise interference to ensure correct matching.

Parameter Settings of the RANSAC Algorithm

- **Distance Threshold:** The RANSAC distance threshold determines which points are considered inliers. A threshold that is too small may exclude many correct matches, while a threshold that is too large may include more incorrect matches, reducing the accuracy of registration.
- **Maximum Iterations:** The maximum number of iterations in RANSAC affects the probability of finding the optimal model. Insufficient iterations may miss the best model, while too many iterations will increase computational costs.

Changes in Viewing Angles and Overlapping Area Size Between Images

- **Viewing Angle Changes:** Larger changes in viewing angles lead to greater differences in the appearance of corresponding points, making matching more challenging. When the viewing angle changes beyond a certain range, the homography model may not accurately describe the transformation between images.
- **Size of the Overlapping Area:** The size of the overlapping area also affects the registration results. A small overlapping area means a limited number of feature points available for matching, and the coplanar assumption may not hold, leading to inaccurate homography estimation.

Other Factors

- **Lighting Changes:** Changes in lighting can alter the appearance of images, affecting feature extraction and matching.
- **Image Distortion:** If image distortions (such as lens distortion) are not well corrected, they can cause registration errors.
- **Low-Texture or Repetitive-Texture Areas:** In areas with low texture or repetitive texture, feature points may be difficult to accurately locate and match.

2.7.2 Experimental Analysis

Ground Truth:

Ginput was use here to get the points manually selected and then used to calculate the ground truth of H.

Below in Figure 2-9 is the code used in this section. We manually choose 7 points from the image and use them to get the H.

```

# ===== task 2.7 =====
# BGR 2 RGB
img1_rgb = cv2.cvtColor(img1, cv2.COLOR_BGR2RGB)
img2_rgb = cv2.cvtColor(img2, cv2.COLOR_BGR2RGB)
# show images
plt.figure()
plt.subplot(121), plt.imshow(img1_rgb), plt.title('Image 1')
plt.subplot(122), plt.imshow(img2_rgb), plt.title('Image 2')
plt.show()
# Select points from Image 1
plt.imshow(img1_rgb)
points1 = plt.ginput(n=-1, timeout=0) # Select points, press Enter to finish
plt.close()
# Select points from Image 2
plt.imshow(img2_rgb)
points2 = plt.ginput(n=-1, timeout=0)
plt.close()

if len(points1) >= 4 and len(points2) >= 4:
    points1 = np.array(points1)
    points2 = np.array(points2)

    # Save points to file
    np.save(file='npfiles/manual_points1.npy', points1)
    np.save(file='npfiles/manual_points2.npy', points2)
else:
    print("At least 4 pairs of points are required.")

u1, v1 = points1[:, 0], points1[:, 1]
u2, v2 = points2[:, 0], points2[:, 1]
H = homography_w_normalisation_ransac(u1, v1, u2, v2)
print(f"Ground truth by manually select points : \n {H}")

```

Figure 2-9 code for task 2-7

Using these points, we get our “ground truth” H in Figure 2-10 and the result stitched image using this H in Figure 2-11 below. Compared with the result in Figure 2-8, we can see that our algorithm get a very familiar H with the “ground truth”.

```

Result H
[[ 6.19895771e-01  1.55743125e-01  2.58991305e+02]
 [-2.89457234e-01  9.01643948e-01  4.85571777e+01]
 [-7.21312955e-04 -1.30689785e-05  1.00000000e+00]]
Ground truth by manually select points :
[[ 5.77947539e-01  2.11697447e-01  2.57707789e+02]
 [-3.12194995e-01  9.41906928e-01  4.65548222e+01]
 [-8.15329364e-04  9.37680268e-05  1.00000000e+00]]

```

Figure 2-10 H from manually choose points

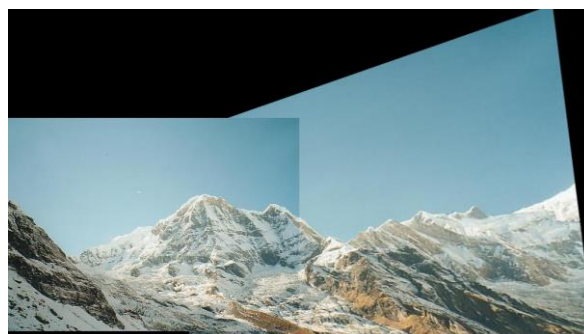


Figure 2-11 result using "ground truth"

This picture in Figure 2-12 is the points selected using ginput on image1.



Figure 2-12 points selected manually in image1

Threshold:

A threshold too low or too high will cause inaccuracy because low thresholds will exclude many correct matches, while high thresholds will include many errors.

The impacts on result were listed below in Table 2-1.

Threshold	0.05	1	20
Result			
Max inliers	7	95	134

Table 2-1 when iterations = 1000

Iterations

Number of iterations decide the possibility of getting best model. Insufficient iterations could miss the best model. A big number of iterations may increase time cost.

The impacts on result were listed below in Table 2-2.




Iterations	500	1000	2000
Result			
Max inliers	87	95	109

Table 2-2 when threshold = 1000

END