Assignment 3

Yifan Luo, u7351505

Task 1: Model Fitting with the Hough Circle Transformation

1. Read RGB images and convert them into greyscale

Assignment_3 - task1.ipynb

```python
# read RGB/RGBA images with the format (H, W, C)
smarties_rgb = cv2.imread('Task1/images/smarties.png')  # (356, 413, 3)
coins_rgba = cv2.imread('Task1/images/coins.png')        # (246, 300, 4)

# convert to grayscale images
smarties = cv2.cvtColor(smarties_rgb, cv2.COLOR_RGB2GRAY)  # (356, 413)
coins = cv2.cvtColor(coins_rgba, cv2.COLOR_RGB2GRAY)        # (246, 300)

# show grayscale images
fig, ax = plt.subplots(1, 2, figsize=(12, 8))
ax[0].imshow(smarties, cmap='gray')
ax[0].set_title(f"smarties\n{smarties.shape}")
ax[0].set_axis_off()
ax[1].imshow(coins, cmap='gray')
ax[1].set_title(f"coins\n{coins.shape}")
ax[1].set_axis_off()
```

Figure 1 Read and convert images

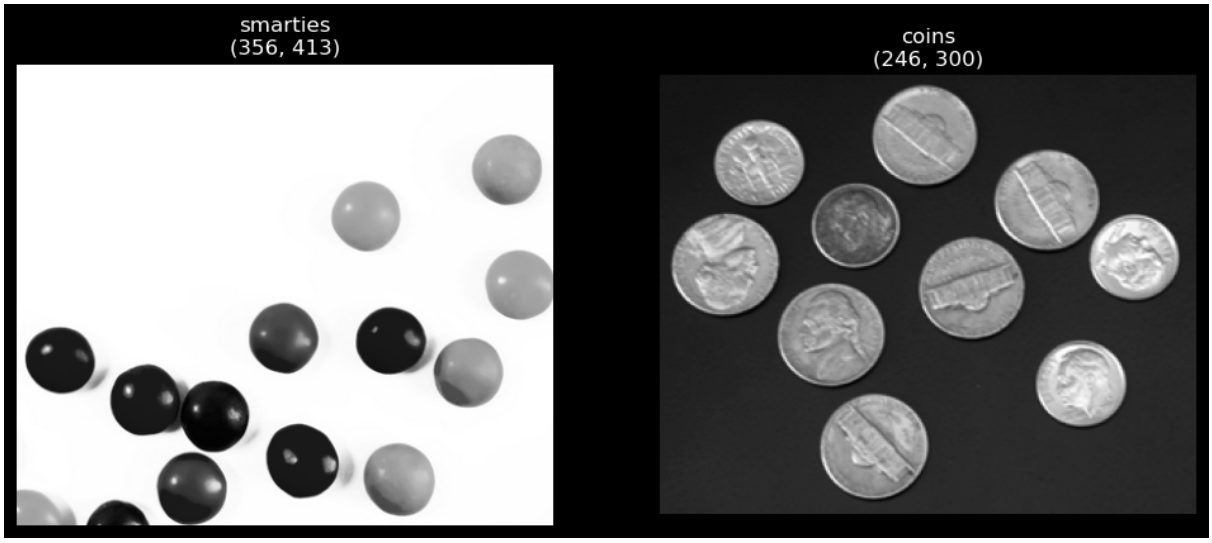

Figure 2 Grayscale images for Task 1

2. Apply the Canny edge detector

Assignment_3 - task1.ipynb

```
# apply Canny edge detecotrs on grayscale images
dge_smarties = cv2.Canny(smarties, threshold1=230, threshold2=320)
edge_coins = cv2.Canny(coins, threshold1=300, threshold2=600)

fig, ax = plt.subplots(1, 2, figsize=(12, 8))
ax[0].imshow(edge_smarties, cmap='gray')
ax[0].set_title(f"edges on smarties\n{edge_smarties.shape}")
ax[0].set_axis_off()
ax[1].imshow(edge_coins, cmap='gray')
ax[1].set_title(f"edges on coins\n{edge_coins.shape}")
ax[1].set_axis_off()
```

Figure 3 Canny edge detection

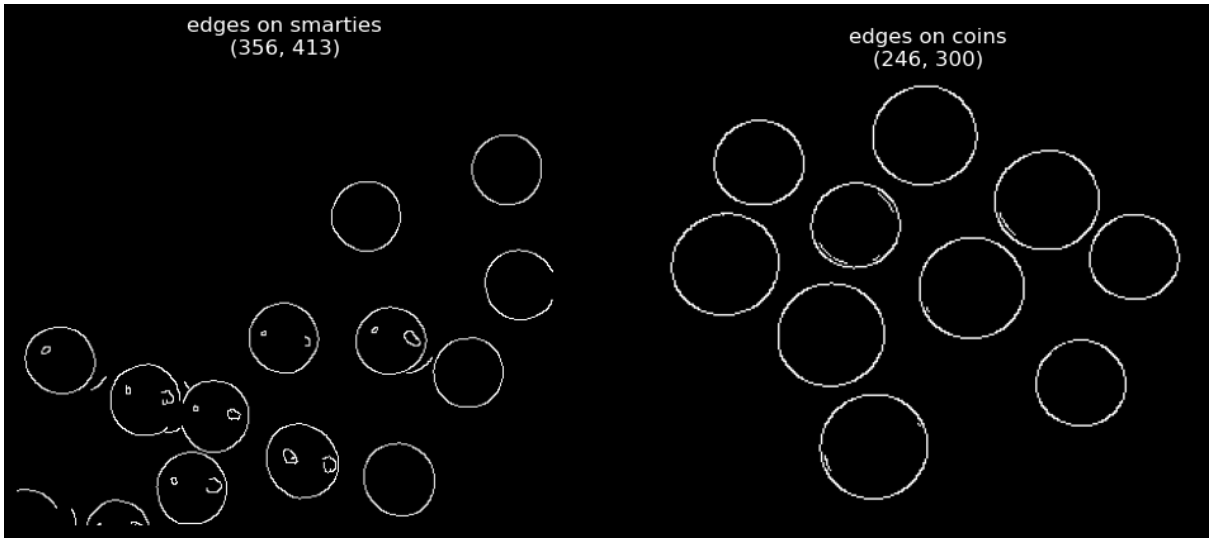

Figure 4 Results from Canny edge detectors

The screenshot above shows the “best” Canny edge detection results after trying multiple threshold combinations.

3. Implement the Hough transform function

Assignment_3 - task1.ipynb

```python
def hough_circle_transform(
    edge,
    threshold_ratio=0.8,
    radius_increment=1,
    circle_centre_increment=[1, 1],
    theta_increment=(2 * np.pi / 180),
    radius_range=[5, 35],
    non_max_suppression=True,
    *args,
    **kwargs
):
    """Implementation of the Hough transform function with non-maximum suppression.

    Args:
        edge (np.ndarray): The Canny edge detection result.
        threshold_ratio (float, optional): Filter small votes. Defaults to 0.8.
        radius_increment (int, optional): The increment of radius during. Defaults to 1.
        circle_centre_increment (list, optional): The increment of the circle centre. Defaults to [1, 1].
        theta_increment (tuple, optional): The increment of theta during. Defaults to (2 * np.pi / 180).
        radius_range (list, optional): The preset radius range. Defaults to [5, 35].
        non_max_suppression (bool, optional): Whether to apply non-maximum suppression. Defaults to True.

    Returns:
        List[int, int, int]: A list of votes [a, b, r] in descending order.
    """
    # initialise a matrix for recording votes
    a_range = int(edge.shape[1] + 2 * radius_range[1] // circle_centre_increment[0])  # padding
    b_range = int(edge.shape[0] + 2 * radius_range[1]// circle_centre_increment[1])   # padding
    r_range = int((radius_range[1] - radius_range[0]) // radius_increment)
    vote = np.zeros((a_range, b_range, r_range))
    # iterate over the edge image and accumulate votes
    for x in range(0, edge.shape[1], circle_centre_increment[0]):
        for y in range(0, edge.shape[0], circle_centre_increment[1]):
            # skip if no edge detected
            if edge[y, x] == 0:
                continue
            # iterate over all the possible theta
            for theta in np.arange(0, 2 * np.pi, theta_increment):
                # iterate over all the possible radius within the range
                for radius in np.arange(radius_range[0], radius_range[1], radius_increment):
                    # a = x - r * cos(θ)
                    candidate_a = x - radius * np.cos(theta)
                    # b = y - r * sin(θ)
                    candidate_b = y - radius * np.sin(theta)
                    # calculate the indices in the vote matrix
                    candidate_a_idx = int((candidate_a // circle_centre_increment[0]) + radius_range[1])
                    candidate_b_idx = int((candidate_b // circle_centre_increment[1]) + radius_range[1])
                    candidate_r_idx = int((radius - radius_range[0]) // radius_increment)
                    # vote if the index is valid
                    if (
                        (0 <= candidate_a_idx < vote.shape[0])
                        and (0 <= candidate_b_idx < vote.shape[1])
                        and (0 <= candidate_r_idx < vote.shape[2])
                    ):
                        vote[candidate_a_idx, candidate_b_idx, candidate_r_idx] += 1

    print(f"after voting: {np.count_nonzero(vote)}")
    # set votes that are lower than the threshold to zeros
    vote = np.where(vote < (threshold_ratio * np.max(vote)), 0, vote)
    print(f"after implying threshold: {np.count_nonzero(vote)}")
    # implement non-maximum suppression
    if non_max_suppression:
        vote = non_maximum_suppression(vote,
                                       kwargs['threshold_a'],
                                       kwargs['threshold_b'],
                                       kwargs['threshold_r'])
        print(f"after non-max suppression: {np.count_nonzero(vote)}")

    # return a list of votes [a, b, r] in a descending order
    asc_idx = np.unravel_index(np.argsort(vote, axis=None), vote.shape)
    res = []
    for a_idx, b_idx, r_idx in zip(asc_idx[0][::-1][:np.count_nonzero(vote)],
                                   asc_idx[1][::-1][:np.count_nonzero(vote)],
                                   asc_idx[2][::-1][:np.count_nonzero(vote)]):
        res.append([(a_idx - radius_range[1]) * circle_centre_increment[0],
                    (b_idx  - radius_range[1]) * circle_centre_increment[1],
                    radius_range[0] + r_idx * radius_increment])
    return np.unique(res, axis=0).tolist()
```

Figure 5 Implementation of the Hough transform function

Assignment_3 - task1.ipynb

```python
def non_maximum_suppression(vote, threshold_a=20, threshold_b=20, threshold_r=20):
    """Implementation of non-maximum suppression.

    Args:
        vote (np.ndarray): The vote matrix.
        threshold_a (int, optional): The half window size in the vertical direction. Defaults to 20.
        threshold_b (int, optional): The hald window size in the horizontal direction. Defaults to 20.
        threshold_r (int, optional): The half window size for radius. Defaults to 20.

    Returns:
        _type_: _description_
    """
    a_len, b_len, r_len = vote.shape
    res = np.zeros_like(vote)
    # zero padding
    vote_pad = np.zeros(
        (a_len + 2 * threshold_a, b_len + 2 * threshold_b, r_len + 2 * threshold_r)
    )
    vote_pad[threshold_a : threshold_a + a_len,
             threshold_b : threshold_b + b_len,
             threshold_r : threshold_r + r_len] = vote
    # non-max suppression
    for a in range(threshold_a, threshold_a + a_len):
        for b in range(threshold_b, threshold_b + b_len):
            for r in range(threshold_r, threshold_r + r_len):
                if vote_pad[a, b, r] == 0:
                    continue
                # the neighbours around [a, b, r]
                # controlled by the three thresholds
                neighbours = vote_pad[
                    a - threshold_a : a + threshold_a + 1,
                    b - threshold_b : b + threshold_b + 1,
                    r - threshold_r : r + threshold_r + 1,
                ]
                # suppression if the vote is not the maximum in its neighbourhood
                if vote_pad[a, b, r] == np.max(neighbours):
                    res[a - threshold_a, b - threshold_b, r - threshold_r] = vote_pad[a, b, r]
    return res
```

Figure 6 The non-maximum suppression in Hough transformation

Figure 5 shows the implementation of the Hough transformation based on the required interface. During the voting stage, in the search space, we only record the corresponding index instead of the real value. Notice that each iteration is controlled by a searching range and an increment.

After applying a threshold and non-maximum suppression, a list of circle candidates is filtered. Before returning the result in descending order, we need to transform the index from the voting matrix into the real [a, b, r] value.

4. Design and implement a non-maximum suppression algorithm

The screenshot is shown in Figure 6 in the last task.

The function takes three thresholds to control the window size of its searching area. Especially, threshold_a (units of pixels) controls the neighbourhood in the vertical direction; threshold_b (units of pixels) controls the neighbourhood in the horizontal

direction threshold_r (units of pixels) controls the searching range of radius. Non-maximum vote will be suppressed to 0 after comparing with its neighbourhood. Only local maxima of the designed neighbourhood are kept.

The function is called inside the hough_circle_transform function shown in Figure 5.

5. Detect circles

Assignment_3 - task1.ipynb

```
# detect coins
circles_coins = hough_circle_transform(
    edge_coins,
    threshold_ratio=0.7,
    radius_increment=1,
    circle_centre_increment=[1, 1],
    theta_increment=(2 * np.pi / 180),
    radius_range=[5, 35],
    non_max_suppression=True,
    threshold_a=20,
    threshold_b=20,
    threshold_r=20
)

fig, ax = plt.subplots()
ax.imshow(coins_rgba)
for x, y, r in circles_coins:
    circle = plt.Circle((x, y), r, color='red', linewidth=2,fill=False)
    ax.add_patch(circle)
    ax.text(x - r + np.random.randint(0, 8),
            y + np.random.randint(-5, 5),
            f"({x}, {y}), {r}",
            color='red', fontsize=10)
ax.set_axis_off()
ax.set_title("coins")
```

Figure 7 Detect circles on coins

```
after voting: 1528133
after implying threshold: 46
after non-max suppression: 11
```

Figure 8 Circles during each stage

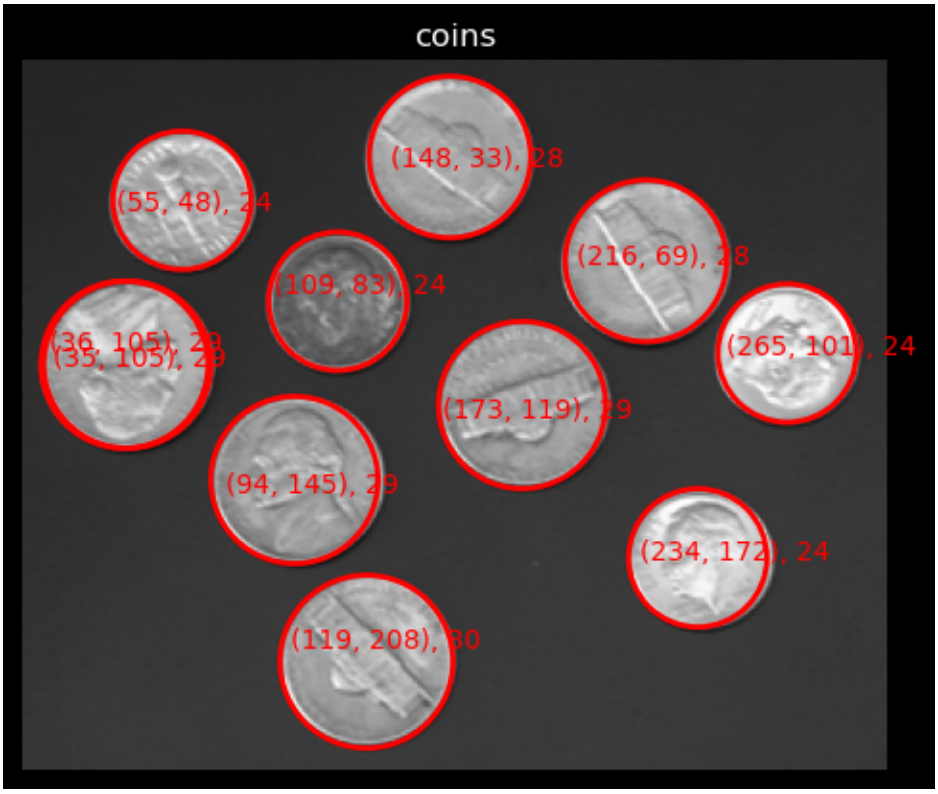

Figure 9 Circles on coins

There are 11 circles detected on the coins. The text over the coins shows the [a, b, r] value for each detection. Two circles are duplicated but are not been suppressed due to they have the same high votes.

Assignment_3 - task1.ipynb

```python
# detect smarties
circles_smarties = hough_circle_transform(
    edge_smarties,
    threshold_ratio=0.7,
    radius_increment=1,
    circle_centre_increment=[1, 1],
    theta_increment=(2 * np.pi / 180),
    radius_range=[10, 35],
    non_max_suppression=True,
    threshold_a=20,
    threshold_b=20,
    threshold_r=20
)

fig, ax = plt.subplots()
ax.imshow(smarties_rgb)
for x, y, r in circles_smarties:
    circle = plt.Circle((x, y), r, color='red', linewidth=2, fill=False)
    ax.add_patch(circle)
    ax.text(x - r + np.random.randint(0, 8),
            y + np.random.randint(-5, 5),
            f"({x}, {y}), {r}",
            color='red', fontsize=8)
    ax.set_title('circles on smarties')
    ax.set_axis_off()
```

Figure 10 Detect circles on smarties

```
after voting: 2070295
after implying threshold: 24
after non-max suppression: 12
```

Figure 11 Detecting results during each stage

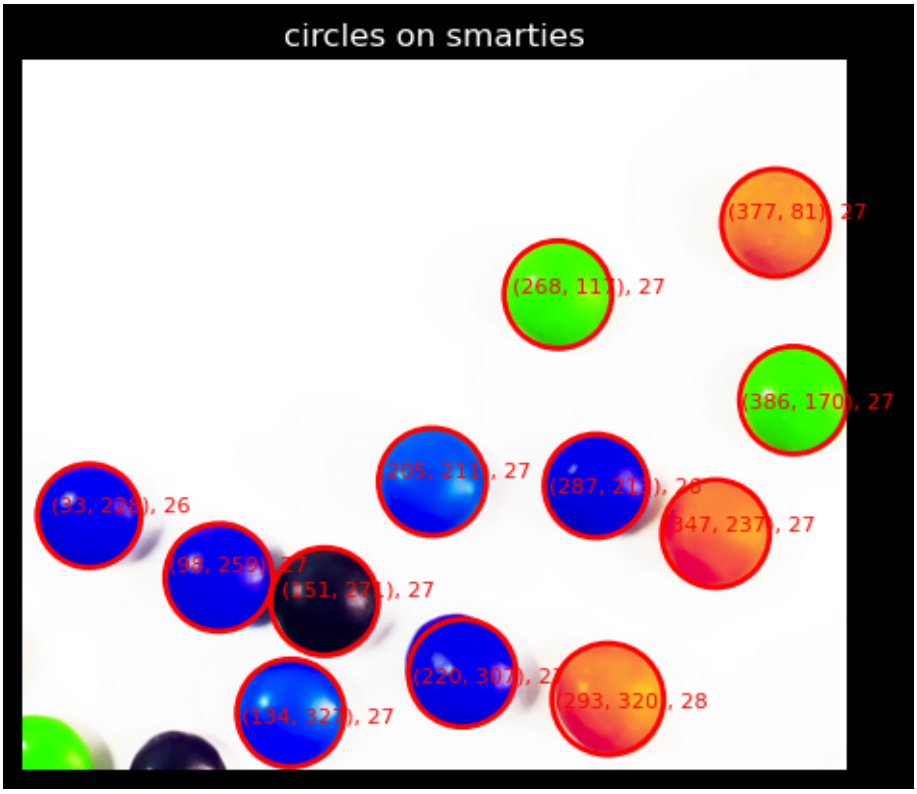

Figure 12 Circles on smarties

Similarly, the above three pictures show the results of smarties. Notice that the bottom-left smarties are not detected due to they have fewer votes and are filtered later by applying the threshold.

6. Analysis the results

For coins:

There are 10 coins in the image as the ground truth. There are 11 circles detected at the end. All ground truths are detected, however, there is a duplicated circle overlayed on the same coin. Hence, FP = 1(the duplicated circle), FNR = 0 (all circles are detected), ACC = 10/11. The reason for the duplicates is that the votes of both circles are the same (89 and 89 shown in the below figure), and they cannot be filtered by applying the threshold or non-maximum suppression.

```
[118. 117. 109. 109. 108. 105. 104. 101.  97.  89.  89.]
```

Figure 13 Top 11 votes in coins

For smarties:

There are 12 full smarties and 2 partial smarties (bottom-left corner) in total as ground truth. The 12 full smarties are all detected, while the 2 partial smarties are not detected. Suppose the 2 partial smarties are also considered as ground truth circles, then FP = 0

(all detected circles are true positive), FNR = 2 / 14 (the bottom-left corner circles are not detected). ACC = 1 (all detected circles are TP).

When using a smaller threshold (more detections) threshold_ratio = 0.1, threshold_a = threshold_b = threshold_r = 5, we can see the bottom-left corner circles can be detected:

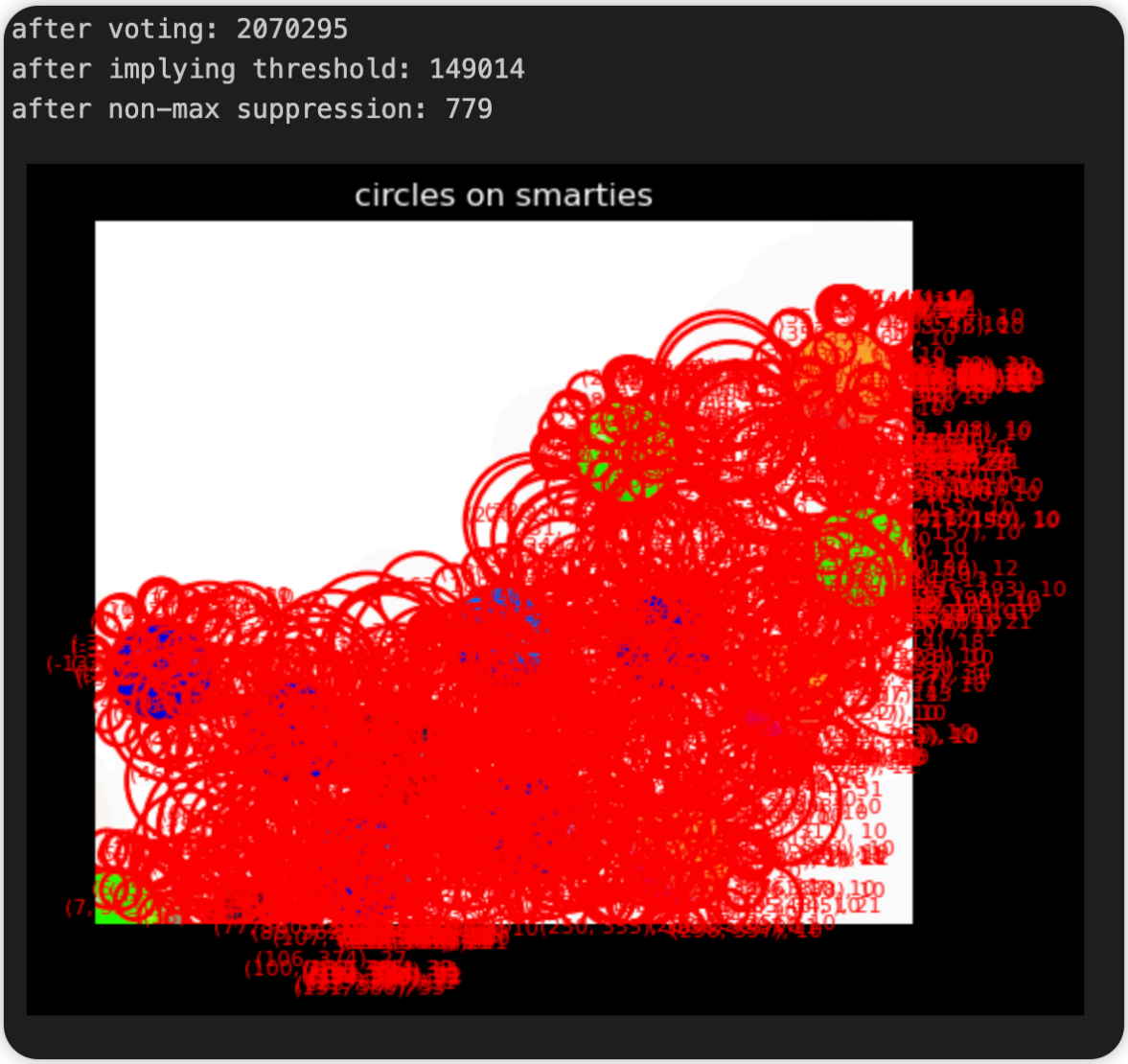

Figure 14 smarties with smaller threshold

After raising threshold_ratio to 0.3 while keeping all other parameters the same, the result shows that only one bottom-left circle is detected, while many false positive circles around other 12 smarlies:

```
after voting: 2070295
after implying threshold: 1496
after non-max suppression: 72
[131. 127. 121. 118. 118. 107. 105. 104. 101.  99.  95.]
```

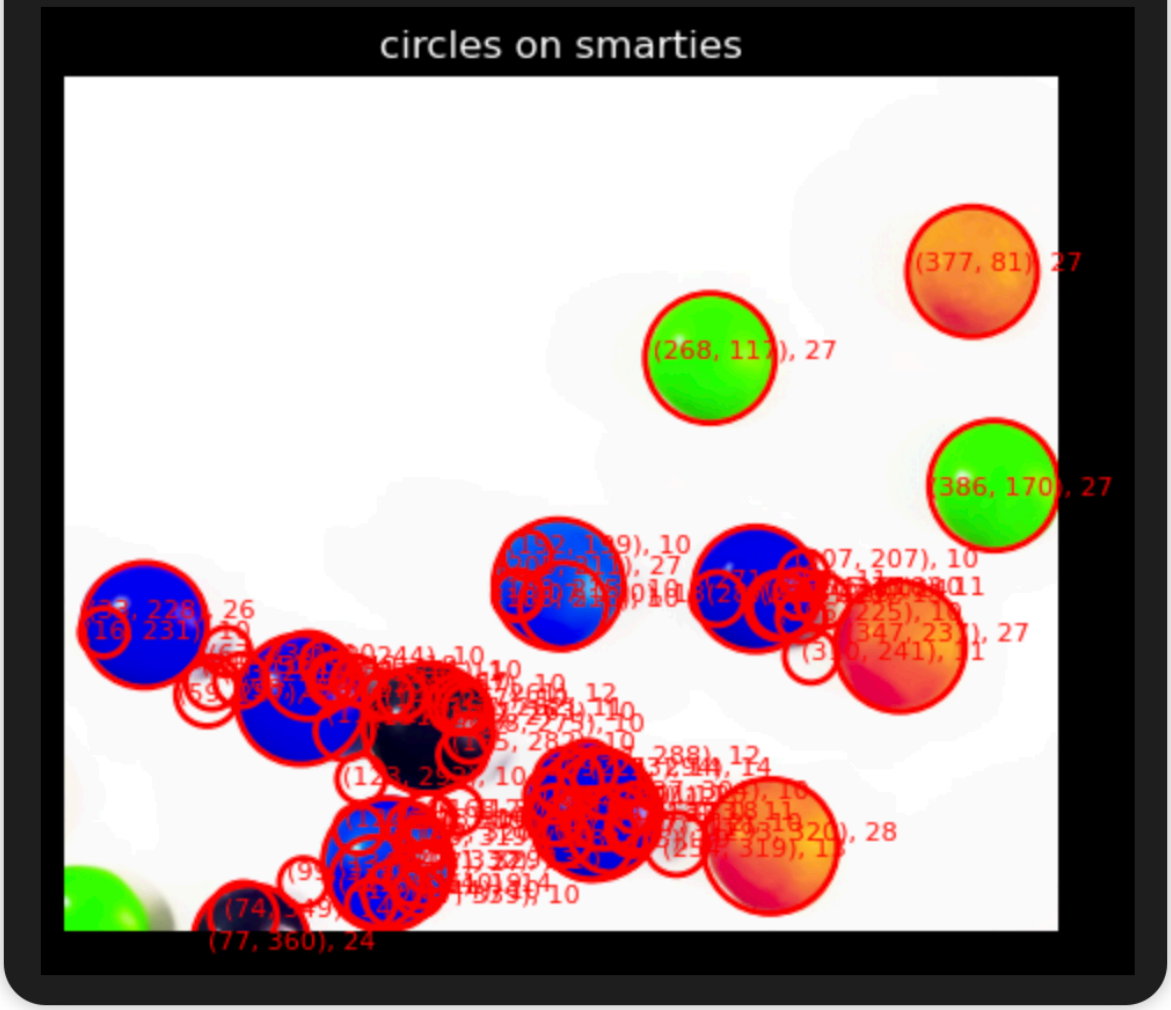

Figure 15 smarlies with larger threshold

When keeping the threshold_ratio as 0.1, and enlarge the threshold_a = threshold_b = 30 and threshold_r = 10, the result is:

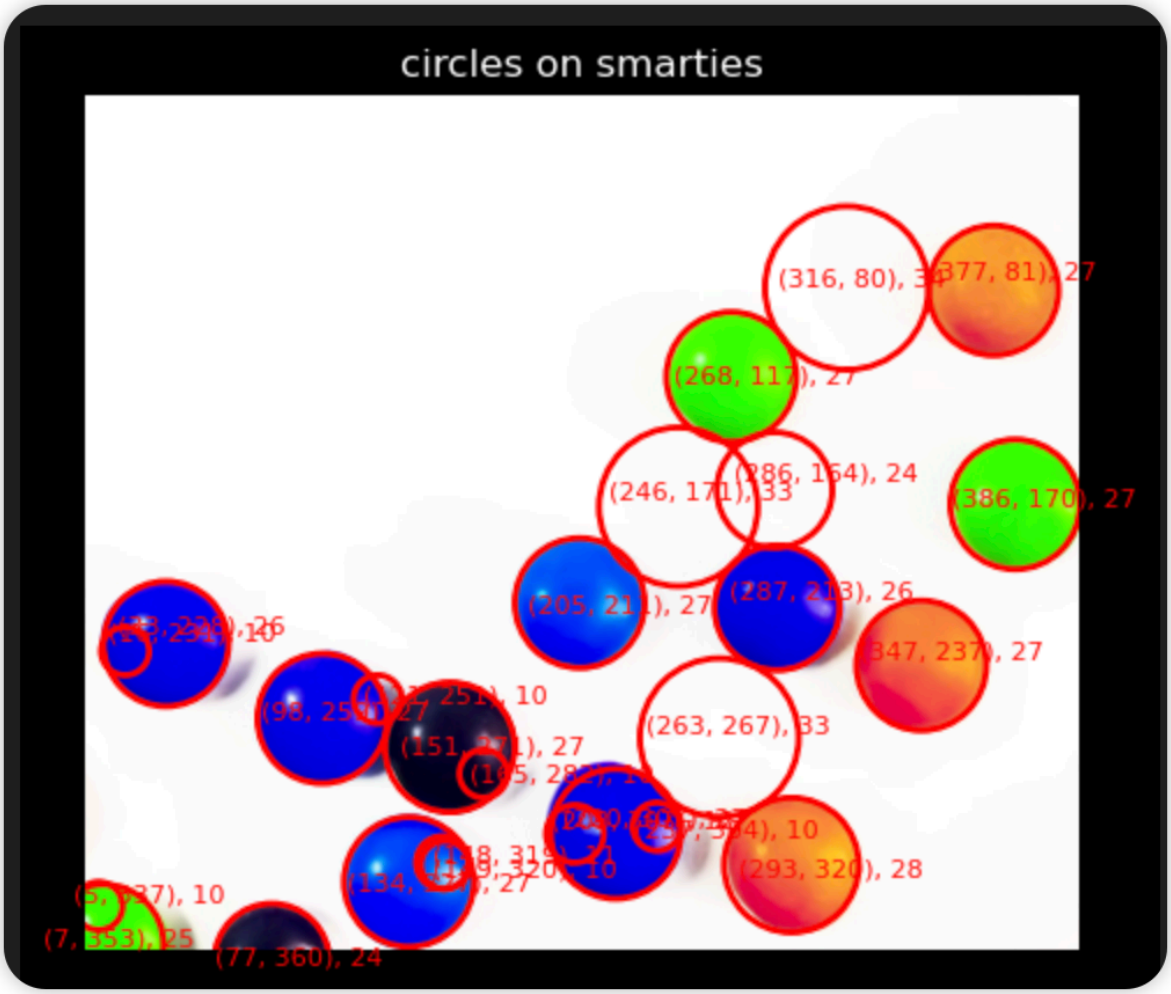

Figure 16 smarlies with large thresholds

Fewer FPs are detected on the 12 smarties, while the bottom-left corner circles are still detected.

Keep enlarging the threshold_r to 50 while keeping others the same, it shows a fairly good result:

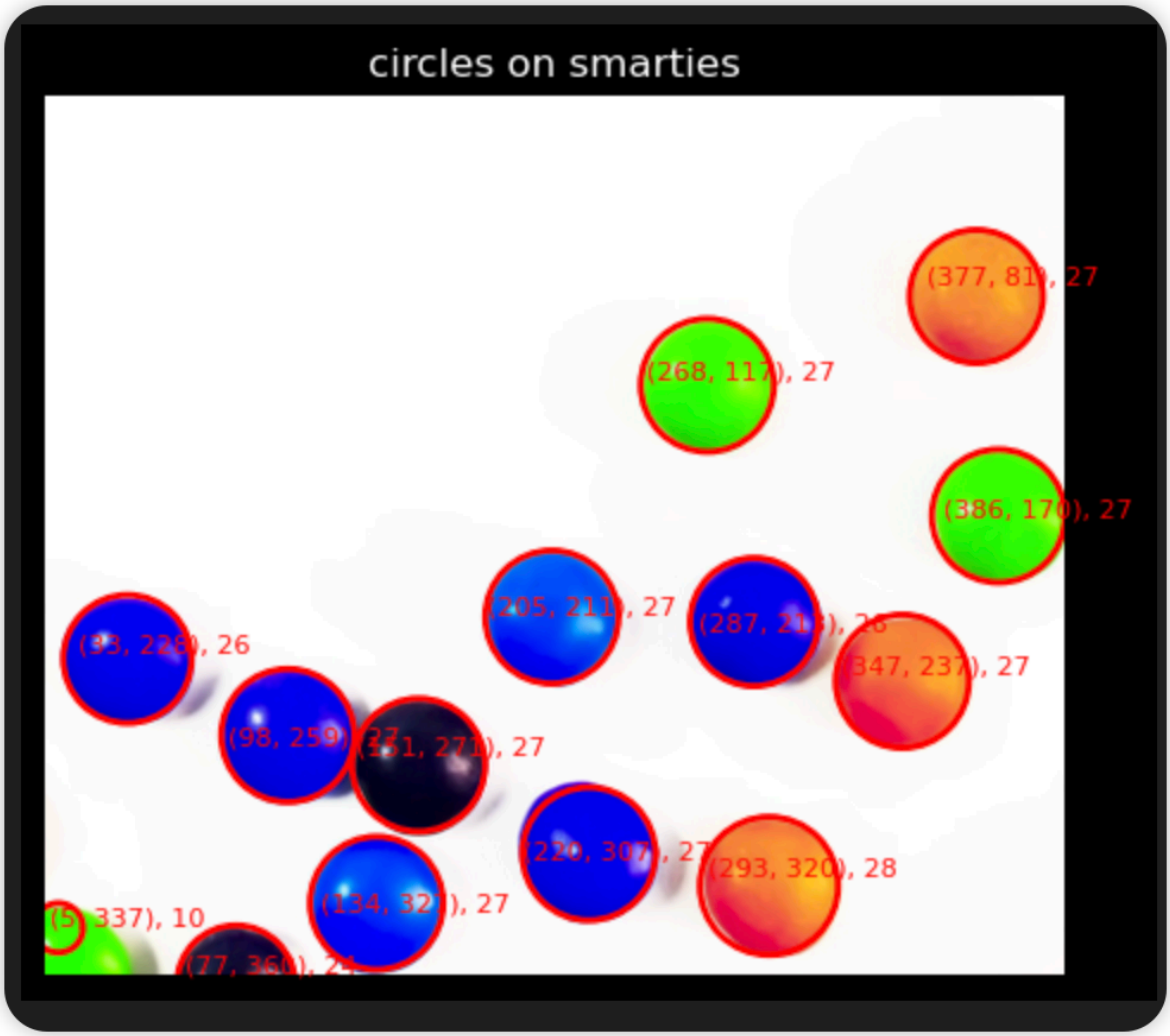

Figure 17 smarties with large thresholds

We can conclude that the bottom-left circles are been filtered by comparing them with the small FP circles overlayed beside them during non-maximum suppression.

From the above analysis and vacuolations, we see that finding a proper set of thresholds is sometimes time-consuming and needs to be tried multiple times to get fairly decent results. The more ground truth the picture has, the more combinations of thresholds we need to try. On the other hand, the complex the pattern is, the more combinations of thresholds we need to try.

7. Examine the impacts of different parameters

Part of the analysis is included in the last question. In this question, we focus on other parameters inside the Hough transformation.

For coins, when setting a large threshould_ratio (0.9) without non-maximum suppression, while setting other increments as 1 unit of pixel and radius range [5, 35], fewer circles are detected:

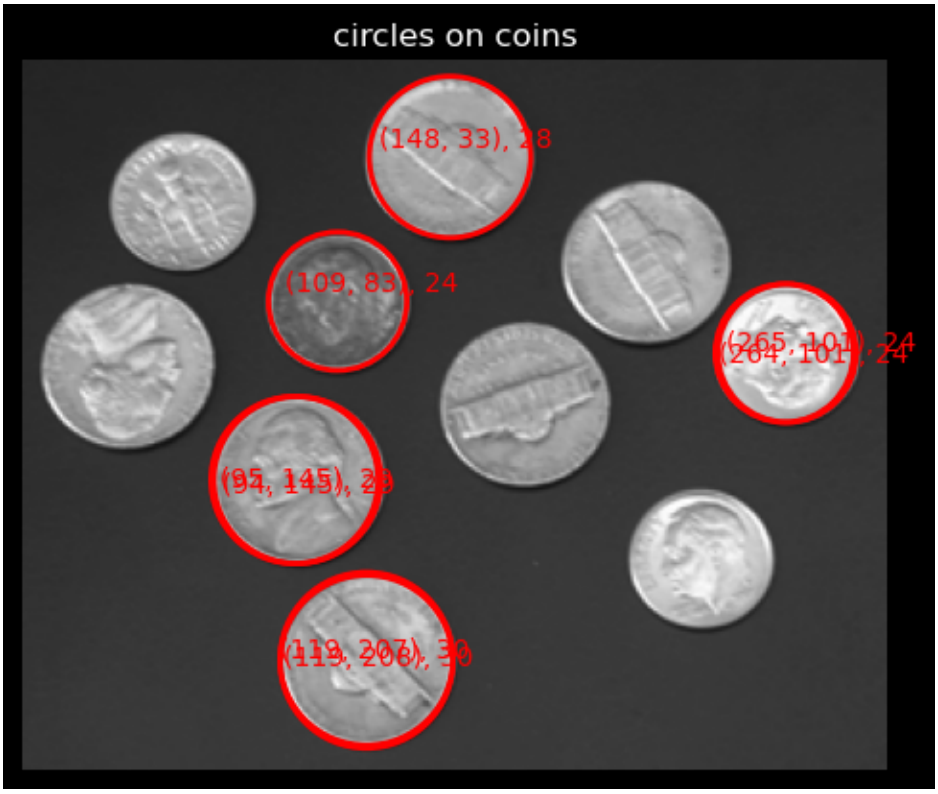

Figure 18 coins with threshold_ratio 0.9

After changing threshold_ratio to 0.4 while keeping others the same, more circles are detected:

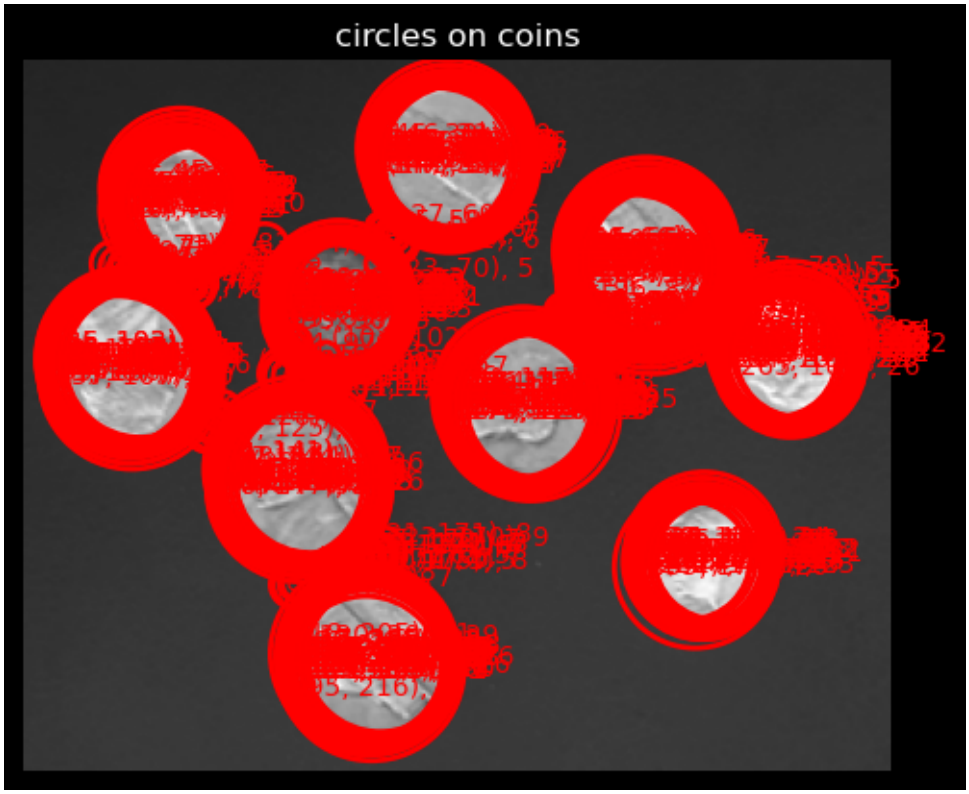

Figure 19 coins with threshold 0.4

We can easily conclude that, when threshold_ratio is set to a large number, only a few circles are left after applying the threshold. Some TP may be left because of improper threshold_ratio. Hence it is better to set it to a moderately small/large number to ensure it can select all TPs.

Now set threshold_ratio to 0.7, and all increments to 1:

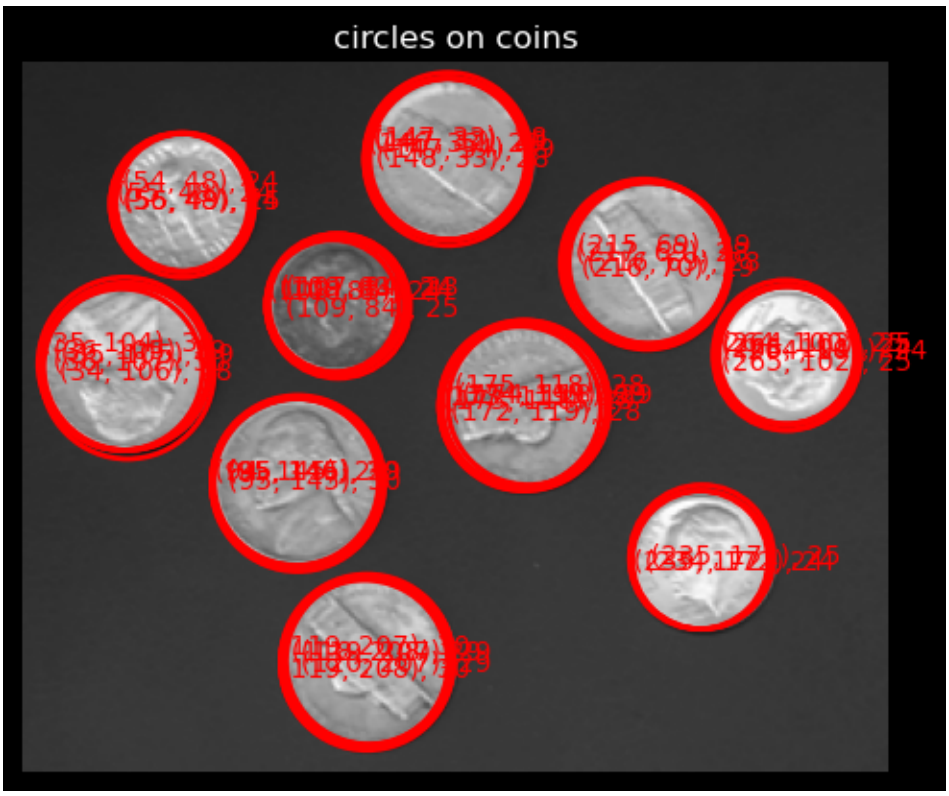

Figure 20 Coins with radius increment 1

When enlarging the radius increment to 3 while keeping others the same:

Figure 21 Coins with radius increment 3

Set radius increment to 10:

Figure 22 Coins with radius increment 10

We can conclude that the radius increment has a strong impact on the results. Different values may generate variant results (i.e., high variance). Hence, it is recommended to set it to a small value to detect all the possible TPs, while keep increasing it for a more efficient computation time.

Similarly, set the circle centre increment to [1, 1]:

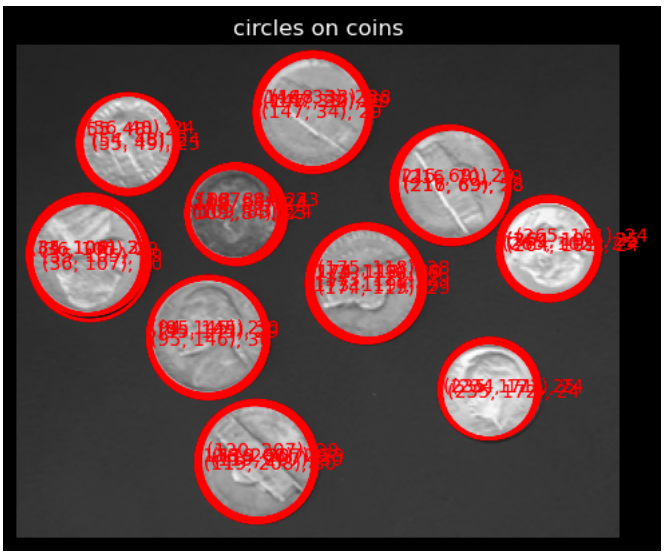

Figure 23 Coins with circle centre increment [1, 1]

Set it to [3, 3]:

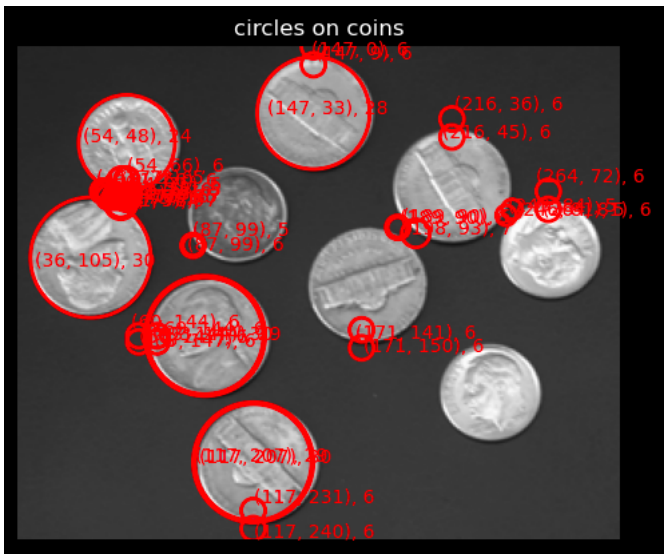

Figure 24 Coins with centre increment [3, 3]

Set it to [6, 6]:

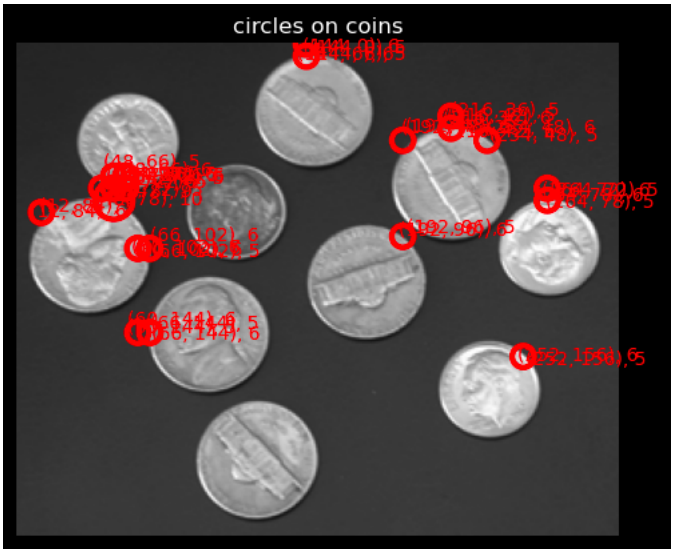

Figure 25 Coins with circle centre increment [6, 6]

We see that fewer circles are detected since the parameter space is getting smaller. It is recommended to set the value to a properly smaller value and increase it gradually to save more computation time.

When setting theta increment to 5*(2*pi/180):

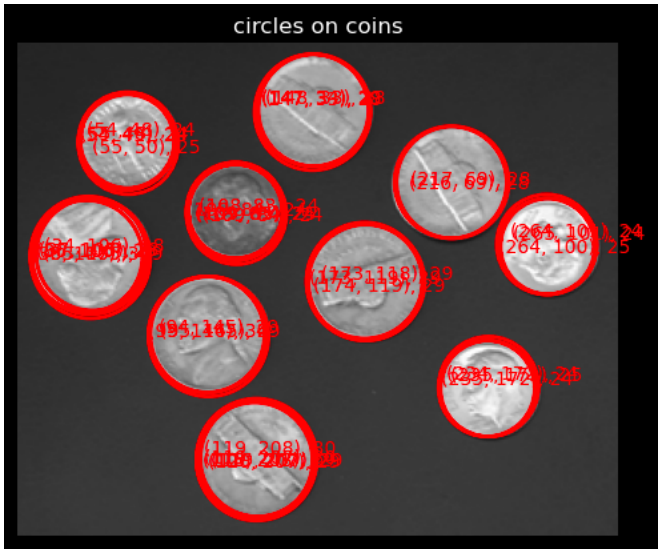

Figure 26 Coins with theta increment to 5(2*pi/180)*

Set it to 10*(2*pi/180):

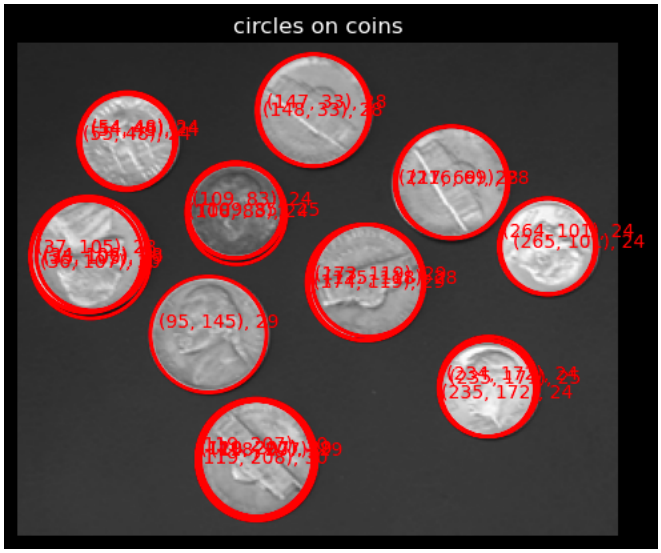

Figure 27 Coins with theta increment to 10(2*pi/180)*

We can see fewer circles are detected since the parameter space is getting smaller. . It is recommended to set the value to a properly smaller value and increase it gradually to save more computation time.

Now set threshold_ratio to 0.3, radius_increment to 5, other increments to 1, and threshold_a = threshold_b = threshold_r = 1:

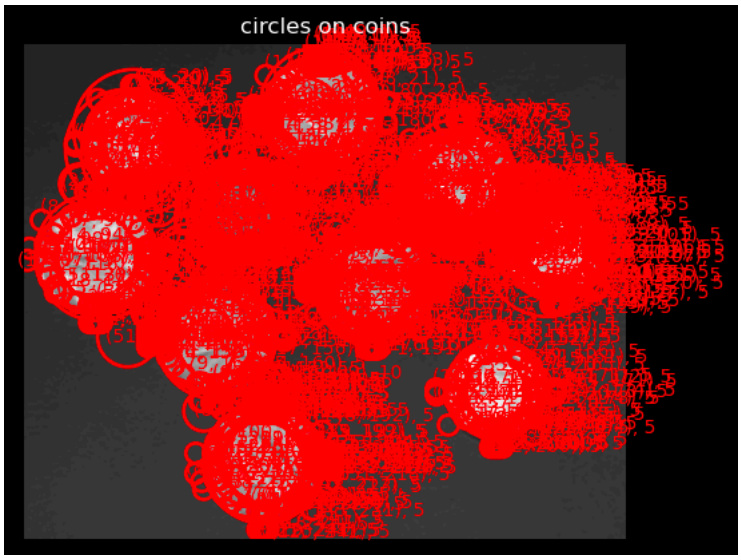

Figure 28 Coins with threshold 1

Set threshold_a = threshold_b = threshold_r = 5:

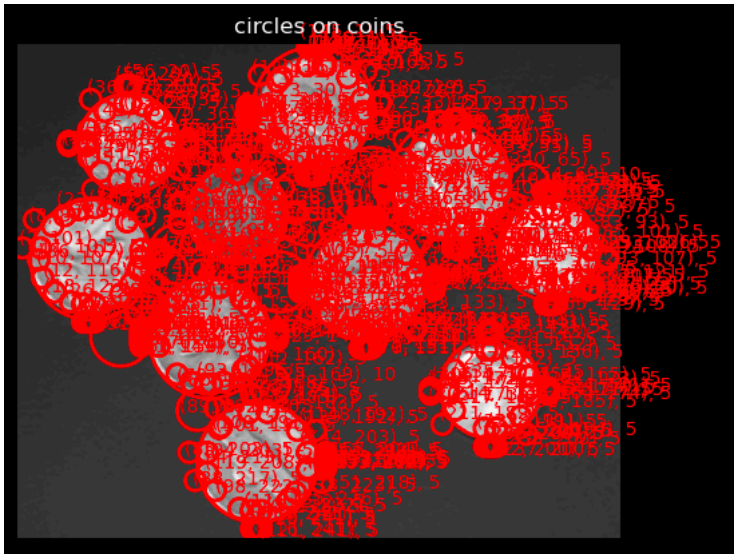

Figure 29 Coins with threshold 5

Set threshold_a = threshold_b = threshold_r = 20:

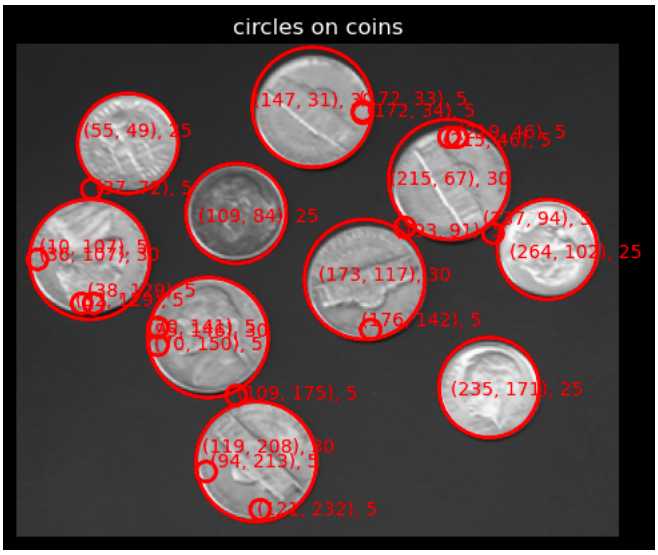

Figure 30 Coins with threshold 20

We can conclude that the larger the threshold, the larger the neighbourhood will be for each non-maximum suppression. It's more recommended to use a large threshold after applying threshold and use non-maximum suppression to filter noises.

8. Report the time and memory usage

For coins, when using threshold_ratio 0.9, radius_increment 1, circle_centre increment [1, 1], theta_increment 2*pi/180, radius_range [5, 35], threshold_a=threshold_b=threshold_r 20, the average running time in Jupyter Notebook is 28.32s.

For non-maximum suppression, the time complexity is O(a*b*r) where a, b, r are the shape of the vote matrix, i.e., the parameter space. The space complexity is also O(a*b*r).

For the Hough circle transformation function, the time complexity is O(H*W*theta*radius), where H and W represent the height and width of the image, and theta and radius represent each search space. The space complexity is O(a*b*r), where a, b, r represent each search space.

To optimise the algorithm:

- Instead of voting for every possible parameter combination, employ strategies like edge gradient-based voting or adaptive voting thresholds to prioritise more promising candidate circles.
- Calculation of different combinations of parameters can be accelerated by parallelisation.
- Using sparse sampling of accumulator space can also speed up the voting process while maintaining robustness.
- Using sparse matrices or compressed representations to store accumulator values and reduce memory usage.
- Techniques like thresholding accumulator values or early stopping criteria based on accumulated evidence can terminate voting for non-promoting circles early.

Task2: Two-view Homography Estimation

1. Read RGB images as NumPy arrays

Figure 31 RGB images in Task 2

2. Implement the homography function

Assignment_3 - task2.ipynb

```python
def homography(u1, v1, u2, v2):
    """Implementation of the DLT algorithm for normalised correspondences.

    Args:
        u1 (np.ndarray or list): The normalised row-wise coordinates of the first image.
        v1 (np.ndarray or list): The normalised column-wise coordinates of the first image.
        u2 (np.ndarray or list): The normalised row-wise coordinates of the second image.
        v2 (np.ndarray or list): The normalised column-wise coordinates of the second image.

    Returns:
        H (np.ndarray): The 3*3 homography matrix that wraps (u1, v1) into (u2, v2).
    """
    # input correspondences should have the same length
    if not len(u1) == len(v1) == len(u2) == len(v2):
        raise ValueError("All input arrays should have the same length")
    # homography requires at least 4 correspondences
    if len(u1) < 4:
        raise ValueError("At least 4 corresponding points are required")
    # the number of total correspondences
    n_correspondences = len(u1)
    # initialise the A matrix where each correspondence contributes two independent rows
    A = np.zeros((2 * n_correspondences, 9))
    # construct the A matrix for each correspondence
    for i in range(n_correspondences):
        A[2 * i] = [0, 0, 0, -u1[i], -v1[i], -1, v2[i] * u1[i], v2[i] * v1[i], v2[i]]
        A[2 * i + 1] = [u1[i], v1[i], 1, 0, 0, 0, -u2[i] * u1[i], -u2[i] * v1[i], -u2[i]]
    # use SVD to decompose the A matrix
    U, S, V_t = np.linalg.svd(A)
    # take the rightmost column of V as the length-9 vector h
    h = V_t.T[:, -1]
    # fix the norm of h to 1
    h /= np.linalg.norm(h)
    # reshape (9,) vector h to a (3, 3) matrix H
    H = h.reshape(3, 3)
    # scale the bottom-right element to 1
    H /= H[-1, -1]
    return H
```

Figure 32 The homography function based on the described interface

The function takes normalised correspondences from both images and conducts an SVD decomposition on the constructed matrix A from all correspondences.

The returned homography matrix H is constructed from the last column of the decomposed matrix V, scaled its norm to 1. Notice that, according to conventions, H is divided by the bottom-right corner element to ensure its value equals 1.

3. Implement the compute_normalisation_matrix function and the homography_w_normalisation function

Assignment_3 - task2.ipynb

```python
def compute_normalisation_matrixs(img1, img2):
    """Generate normalisation matrices for input images.

    Args:
        img1 (np.ndarray): The image to calculate the normalisation matrix T.
        img2 (np.ndarray): The image to calculate the normalisation matrix T'.

    Returns:
        T, T' (tuple(np.ndarray, np.ndarray)): The normalisation matrices for img1 and img2.
    """
    # the height and width for both images
    w1, h1 = img1.shape[1], img1.shape[0]
    w2, h2 = img2.shape[1], img2.shape[0]
    # calculate the normalisation matrix T and T'
    T1 = np.linalg.inv(np.array([[w1 + h1, 0, w1 / 2],
                                 [0, w1 + h1, h1 / 2],
                                 [0, 0, 1]]))
    T2 = np.linalg.inv(np.array([[w2 + h2, 0, w2 / 2],
                                 [0, w2 + h2, h2 / 2],
                                 [0, 0, 1]]))
    return T1, T2
```

Figure 33 The compute_normalisation_matrix for computing normalisation matrices from input pictures

The screenshot above shows the implementation of calculating normalisation matrices T and T' based on slides. The function returns calculated T and T' for later use.

Figure 34 The homography_w_normalisation function to calculate unnormalised homography matrix H

The function above first calculates the normalisation matrices T1 and T2 by using the previous function compute_normalisation_matrix. Then, the unnormalised coordinates u1, v1, u2 and v2 are constructed and transformed into normalised coordinates using T1 and T2.

The homography function is used to calculate homography matrix with normalised coordinates u1_norm, v1_norm, u2_norm and v2_norm. Then, the homography matrix is denormalised by using T1 and T2 and scaled by using the bottom-right corner entry.

4. Use SIFT keypoint detectors and feature descriptors to find correspondences

Assignment_3 - task2.ipynb

```python
# initialise the SIFT feature detector
sift = cv2.SIFT_create()

mountain1_bgr = cv2.imread('Task2/images/mountain1.jpg')
mountain2_bgr = cv2.imread('Task2/images/mountain2.jpg')

# detect keypoints and compute descriptors
kp1, des1 = sift.detectAndCompute(mountain1_gray, None)
kp2, des2 = sift.detectAndCompute(mountain2_gray, None)

# draw the keypoints
mountain1_sift = cv2.drawKeypoints(mountain1_gray, kp1, mountain1_bgr, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
mountain2_sift = cv2.drawKeypoints(mountain2_gray, kp2, mountain2_bgr, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
fig, ax = plt.subplots(1, 2, figsize=(10, 8))
ax[0].imshow(mountain1_sift)
ax[0].set_axis_off()
ax[0].set_title(f"mountain1.jpg\n{len(kp1)} keypoints")
ax[1].imshow(mountain2_sift)
ax[1].set_axis_off()
ax[1].set_title(f"mountain2.jpg\n{len(kp2)} keypoints")
plt.show()
```

Figure 35 Use SIFT to find keypoints and descriptors

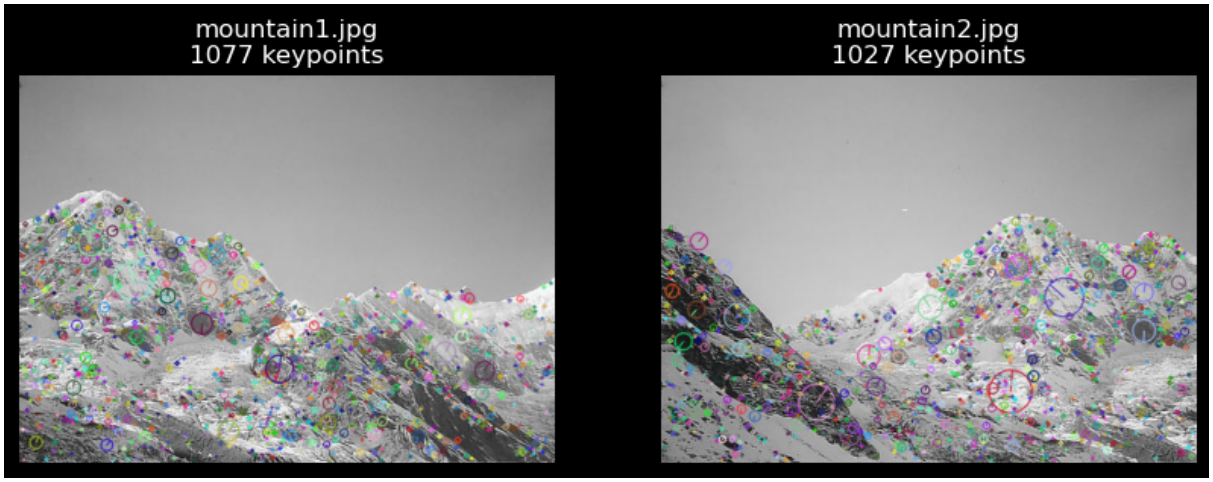

Figure 36 The detected keypoints from both images

First, read the original images in BGR and grayscale formats. Then, use cv2.SIFT_create() to create SIFT detectors and find keypoints (kp1, kp2) and descriptors (des1, des2) for mountain1 and mountain2. The results show mountain1 has 1077 keypoints and mountain2 has 1027 keypoints.

Assignment_3 - task2.ipynb

```python
# reference: https://docs.opencv.org/4.x/d5/d6f/tutorial_feature_flann_matcher.html

# find matches by using FLANN
matcher = cv2.DescriptorMatcher_create(cv2.DescriptorMatcher_FLANNBASED)
knn_matches = matcher.knnMatch(des1, des2, 2)

# filter matches using the Lowe's ratio test
# the lower the ratio, the fewer the correspondences
ratio = 0.5
all_matches = []
for m, n in knn_matches:
    if m.distance < ratio * n.distance:
        all_matches.append(m)
```

Figure 37 Match and select correspondences with a threshold

The figure above shows how to use a matcher to match keypoints in OpenCV. Notice that we use a ratio of 0.5 as a threshold to select from all the correspondences. The lower the ratio, the fewer the matches.

Assignment_3 - task2.ipynb

```python
# draw all matches
img_all_matches = np.empty((max(mountain1_gray.shape[0], mountain2_gray.shape[0]),
                            mountain1_gray.shape[1] + mountain2_gray.shape[1], 3),
                           dtype=np.uint8)
cv2.drawMatches(mountain1_gray, kp1, mountain2_gray, kp2, all_matches, img_all_matches, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# sample half of the correspondences
np.random.seed(42)
N = int(np.ceil(len(all_matches) / 2))
sample_idx = np.random.choice(len(all_matches), N, replace=False)
sample_matches = np.asarray(all_matches)[sample_idx]

# draw sampled matches
img_matches = np.empty((max(mountain1_gray.shape[0], mountain2_gray.shape[0]),
                        mountain1_gray.shape[1] + mountain2_gray.shape[1], 3),
                       dtype=np.uint8)
cv2.drawMatches(mountain1_gray, kp1, mountain2_gray, kp2, sample_matches, img_matches, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

fig, ax = plt.subplots(2, 1, figsize=(8, 6))
ax[0].imshow(img_all_matches)
ax[0].set_title(f"All Matches ({len(all_matches)} Points)")
ax[0].set_axis_off()
ax[1].imshow(img_matches)
ax[1].set_title(f"Sampled Matches ({len(sample_matches)} Points)")
ax[1].set_axis_off()
```

Figure 38 Draw all the matches and the selected matches by using a seed 42

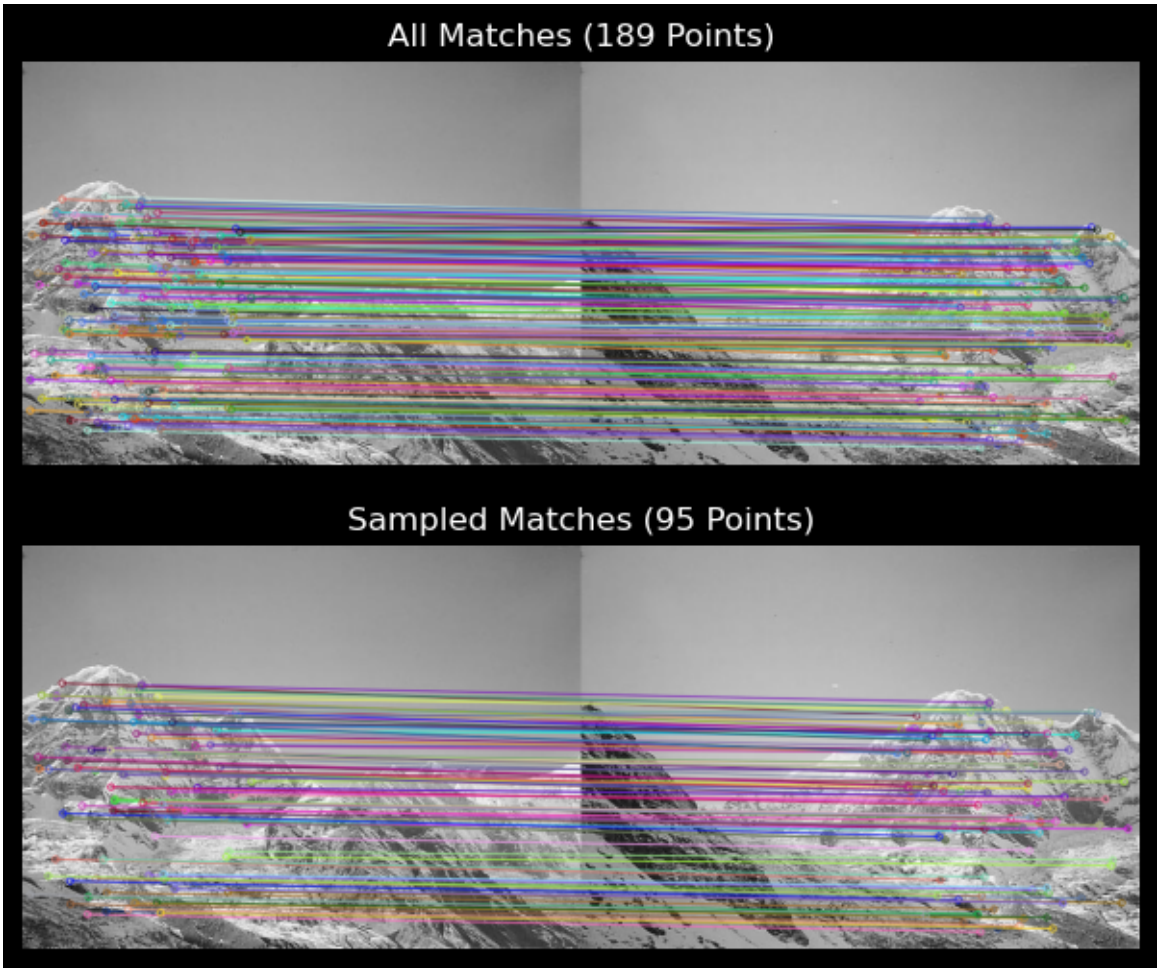

Figure 39 The all matches and the selected matches

There are 189 matches under the ratio of 0.5, and 95 selected matches according to the requirement.

Assignment_3 - task2.ipynb

```
# convert cv2.DMatch to np.array
samples = np.array([[s.queryIdx, s.trainIdx, s.distance] for s in sample_matches])
# save sampled matches as a .npy file
samples_path = 'Task2/sample_matches.npy'
np.save(samples_path, samples)
print(f"Saved into {samples_path}")
# load samples from .npy file
print(f"Loading from {samples_path}")
samples_load = np.load("Task2/sample_matches.npy")
print(samples_load[:5])
```

Figure 40 Save and load from a "npy" file

```
Saved into Task2/sample_matches.npy
Loading from Task2/sample_matches.npy
[[ 539.         1016.          140.10353088]
 [ 477.          915.          102.95144653]
 [  94.          554.          149.14086914]
 [  84.          599.           99.85489655]
 [ 228.          647.           66.61831665]]
```

Figure 41 The NumPy array loaded from the "npy" file

A ".npy" file is saved for the NumPy array object which contains the coordinates of matches from OpenCV. The above picture shows the array information of the first 5 matches. The first column is the query index for mountain1, the second is the target index for mountain2, and the last is the distance of each match.

5. Implement homography_w_normalisation_ransac

Assignment_3 - task2.ipynb

```python
def homograpy_w_normalisation_ransac(img1, kp1, img2, kp2, corr, n_matches,
                                     dist_ratio, max_iter=10, seed=42):
    """ Implementation of using RANSAC to select the best performing H
    that has the most amount of inliners

    Args:
        img1 (np.ndarray): The input query image.
        kp1 (np.ndarray): The keypoints derived from OpenCV in img1.
        img2 (np.ndarray): The input target image.
        kp1 (np.ndarray): The keypoints derived from OpenCV in img2.
        corr (np.ndarray): The matches from img1 and img2, aligned with kp1 and kp2.
        n_matches (int): The number of matches to sample.
        dist_ratio (float): The ratio of distance for selecting inliners.
        max_iter (int): The max iteration in RANSAC.
        seed (int): The NumPy random seed for sampling in RANSAC.

    Returns:
        best_H (np.ndarray): The best performing H that has the most amount of inliners.
    """
    # set the seed for sampling in RANSAC
    np.random.seed(seed)
    # record the maximum number of inliners during RANSAC
    max_inliers = 0
    # record the bast performed H that has the most amount of inliners
    best_H = np.zeros((3, 3))
    for _ in range(max_iter):
        # sample n matches to estimate H
        sample_idx = np.random.choice(len(corr), n_matches, replace=False)
        sample_matches = np.asarray(corr)[sample_idx]
        # construct the coordinates u1, v1, u2, v2 from matches
        u1, v1 = np.array([]), np.array([])
        u2, v2 = np.array([]), np.array([])
        for m in sample_matches:
            v1_, u1_ = kp1[m.queryIdx].pt
            v2_, u2_ = kp2[m.trainIdx].pt
            u1 = np.append(u1, u1_)
            v1 = np.append(v1, v1_)
            u2 = np.append(u2, u2_)
            v2 = np.append(v2, v2_)
        # calculate the homography matrix for unnormalised coordinates
        H = homograpy_w_normalisation(img1, u1, v1, img2, u2, v2)
        # calculate the symmetric transfer error for each correspondence
        X_1 = np.hstack((u1[:, np.newaxis], v1[:, np.newaxis], np.ones((len(u1), 1))))
        X_2 = np.hstack((u2[:, np.newaxis], v2[:, np.newaxis], np.ones((len(u2), 1))))
        # calcute the symmetric transfer error for each match
        sym_trans_err = np.square(X_1.T - np.linalg.inv(H) @ X_2.T).T.sum(axis=1) + \
                        np.square(X_2.T - H @ X_1.T).T.sum(axis=1)
        # calculate the number of inliers that are within dist_ratio * max(dist)
        # record the best H that has the largest number of inliers
        n_inliers = len(sym_trans_err[sym_trans_err < dist_ratio * np.max(sym_trans_err)])
        if n_inliers > max_inliers:
            max_inliers = n_inliers
            best_H = H

    return best_H
```

Figure 42 The homography_w_normalisation_ransac function to select the best performing H using RANSAC

The screenshot above shows the implementation of using RANSAC to select the best H. In each round, the algorithm samples several matches and uses these matches to calculate a homography matrix H. We use symmetric transfer error and a distance ratio for each match to select inliers. Record and return the best-performing H at the end.

Assignment_3 - task2.ipynb

```
n_matches = 15
dist_ratio = 0.08
max_iter = 30
best_H = homograpy_w_normalisation_ransac(mountain1_gray, kp1, mountain2_gray, kp2, sample_matches,
                                          n_matches=n_matches, dist_ratio=dist_ratio, max_iter=max_iter)
print(f"Each H is calculated from {n_matches} matches\n\
Inliers are selected under the distance ratio {dist_ratio}\n\
In the total {max_iter} iterations")
print(f"The best H found by RANSAC:\n{best_H}")
```

Figure 43 Show the best performing H

```
Each H is calculated from 15 matches
Inliers are selected under the distance ratio 0.08
In the total 30 iterations
The best H found by RANSAC:
[[ 8.50960152e-01 -2.66757425e-01  5.21526448e+01]
 [ 1.19510164e-01  6.07816803e-01  2.59509346e+02]
 [-1.33230239e-04 -6.77076891e-04  1.00000000e+00]]
```

Figure 44 The best H found by RANSAC

The result shown above is the best-performing H found by RANSAC. In the total 30 samplings, each H is calculated using 15 sampled matches (set seed to 42), and the liners were selected with a distance (symmetric transfer error) threshold of 0.08.

6. Warp and stitch the results

Assignment_3 - task2.ipynb

```python
def calculate_new_size(img, H):
    """ Calculate the transformed image size.

    Args:
        img (np.ndarray): The input image to be trasnformed.
        H (np.ndarray): The transformation matrix.

    Returns:
        height, width, shift_u, shift_v (tuple(int, int, int, int)): The new size and shift infromation.

    """
    corners = np.array([[0, 0, 1],                        # top left
                        [0, img.shape[1], 1],             # top right
                        [img.shape[0], 0, 1],             # bottom left
                        [img.shape[0], img.shape[1], 1]])  # bottom right
    # transformed coordinates by H
    corners_trans = (H @ corners.T).T  # ((3, 3) @ (4, 3).T).T = (4, 3)
    u2 = corners_trans[:, 0] / corners_trans[:, 2]
    v2 = corners_trans[:, 1] / corners_trans[:, 2]
    # find the new height and width of the transfermed image
    height = np.ceil(u2.max() - u2.min()).astype(int)
    width = np.ceil(v2.max() - v2.min()).astype(int)
    # find the shift in both horizontal and vertical directions
    # to calculate the homography coordinates of the tranformed image
    shift_u = np.ceil(u2.min() - 0).astype(int)
    shift_v = np.ceil(v2.min() - 0).astype(int)
    return height, width, shift_u, shift_v


height, width, shift_u, shift_v = calculate_new_size(mountain1, best_H)
print(f"new image height: {height}, width: {width}")
print(f"shift vertically: {shift_u}, horizontally shift_v: {shift_v}")
```

Figure 45 The function used to calculate the new size and shift information for inverse wrapping

```
new image height: 522, width: 772
shift vertically: -131, horizontally shift_v: 260
```

For mountain1, the transformed image size has a height of 522 and a width of 772. The shift information is used in the later inverse warping with the bilinear interpolation technique.

Assignment_3 - task2.ipynb

```python
def inverse_warping(img, H):
    """Inverse warping for the input image with the transformation matrix.

    Args:
        img (np.ndarray): The input image to be transformed.
        H (np.ndarray): The transformation matrix

    Returns:
        res (np.ndarray): The transformed image.
    """
    # calculate the size and shift information for later use
    new_height, new_width, shift_u, shift_v = calculate_new_size(mountain1, best_H)
    # initialise the transformed image to return
    res = np.zeros((new_height, new_width, 3 if img.ndim == 3 else 1))
    # check if (u, v) is valid in img
    cond = lambda x, y: (0 <= x < img.shape[0]) and (0 <= y < img.shape[1])
    # iterate over the pixels in res
    for u2 in range(new_height):
        for v2 in range(new_width):
            # the coordinate in target image (X_2 = best_H @ X_1)
            X_2 = np.array([u2 + shift_u, v2 + shift_v, 1])
            # the coordinate in the query image (inv(best_H) @ X2 = X1)
            X_1 = np.linalg.inv(H) @ X_2
            # transform the homography coordinates to Cartesian coordinates
            u1 = X_1[0] / X_1[-1]
            v1 = X_1[1] / X_1[-1]
            if cond(u1, v1):
                if int(u1) == u1 and int(v1) == v1:
                    res[u2, v2] = img[int(u1), int(v1)]
                else:
                    # bilinear interpolation
                    x1, x2 = int(np.floor(u1)), int(np.ceil(u1))
                    y1, y2 = int(np.floor(v1)), int(np.ceil(v1))
                    if cond(x1, y2) and cond(x2, y2) and cond(x1, y1) and cond(x2, y1):
                        if x1 == x2:
                            res[u2, v2] = img[x1, y2] * (y - y1) / (y2 - y1) + img[x1, x1] * (y2 - y) / (y2 - y1)
                        elif y1 == y2:
                            res[u2, v2] = img[x2, y2] * (x2 - x) / (x2 - x1) + img[x1, y2] * (x - x1) / (x2 - x1)
                        else:
                            R1 = img[x1, y2] * ((x2 - u1) / (x2 - x1)) + img[x2, y2] * ((u1 - x1) / (x2 - x1))
                            R2 = img[x1, y1] * ((x2 - u1) / (x2 - x1)) + img[x2, y1] * ((u1 - x1) / (x2 - x1))
                            res[u2, v2] = R1 * (v1 - y1) / (y2 - y1) + R2 * (y2 - v1) / (y2 - y1)

    return res.astype('uint8')

mountain1_trans = inverse_warping(mountain1, best_H)

# plot the trasnformed result
fig, ax = plt.subplots()
ax.imshow(mountain1_trans)
ax.set_title(f"Trasnformed mountain1\n{mountain1_trans.shape}")
ax.set_axis_off()
```

Figure 46 Inverse warping for mountain1

Figure 47 The transformed mountain1

Assignment_3 - task2.ipynb

```
# scale the transformed mountain1
ratio = 0.89
mountain1_resize = cv2.resize(mountain1_trans, (int(mountain1_trans.shape[1] * ratio),
                                                int(mountain1_trans.shape[0] * ratio)))

# stitch images (left: mountain2, right: mountain1)
stitched_img = np.zeros((374, mountain2.shape[1] + mountain1_resize.shape[1], 3)).astype('uint8')
mountain2_cut = 313
mountain1_cut = 70
stitched_img[:, :mountain2_cut] = mountain2[:, :mountain2_cut]
stitched_img[:, mountain2_cut:(mountain2_cut + (mountain1_resize.shape[1] - mountain1_cut))] = mountain1_resize[mountain1_resize.shape[0] - 374:, mountain1_cut:]

plt.figure(figsize=(12, 8))
plt.title(f"The Stitched Result\n{stitched_img.shape}")
plt.imshow(stitched_img[:, :mountain2_cut + (mountain1_resize.shape[1] - mountain1_cut)])
plt.axis('off')
plt.show()
```

Figure 48 Stitch two images

The stitching process includes scaling for mountain2. The stitch points from both images are selected based on the visual results from the previous SIFT keypoints and descriptors.

Figure 49 The stitch result (left: mountain2, right: mountain1)

7. Discussion

First, we use ginput() to select 8 correspondences from both images and save the correspondence coordinates into “point_1.npy” for mountain1 and “point_2.npy” for mountain2.

Assignment_3 - select_points.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  path = 'Task2/images/mountain2.jpg'
5  save_path = 'Task2/mountain2_point.npy'
6  # Load an image
7  mountain = plt.imread(path)
8  # Display the image
9  plt.imshow(mountain)
10 # Use plt.ginput() to select points
11 selected_points = plt.ginput(n=8, timeout=0)
12 # Close the plot
13 plt.close()
14 # Print the selected points
15 print("Selected points:", selected_points)
16 np.save(save_path, selected_points)
```

Figure 50 Select 8 correspondences

The manually selected correspondences are:

Figure 51The manually selected correspondences

Use homography_w_normalisation to calculate the homography matrix:

Assignment_3 - task2.ipynb

```
# calculate the homography matrix with the manually selected correspondeces
H_select = homograpy_w_normalisation(mountain1, points_1[:, 0], points_1[:, 1],
                                     mountain2, points_2[:, 0], points_2[:, 1])
print(H_select)
```

Figure 52 The homography matrix calculated with the 8 correspondences

```
[[ 6.37507672e-01  1.00711146e-01  2.59295683e+02]
 [-2.67214379e-01  8.32356450e-01  5.36026067e+01]
 [-6.17521544e-04 -1.86741367e-04  1.00000000e+00]]
```

Figure 53 The calculated homography matrix

Compared with the results (RANSAC) in Task 2.5, some entries have significant differences.

The following plot shows the stitch results by using the manually selected correspondences. Notice that we use the similar stitch location as the previous one for a better comparison:

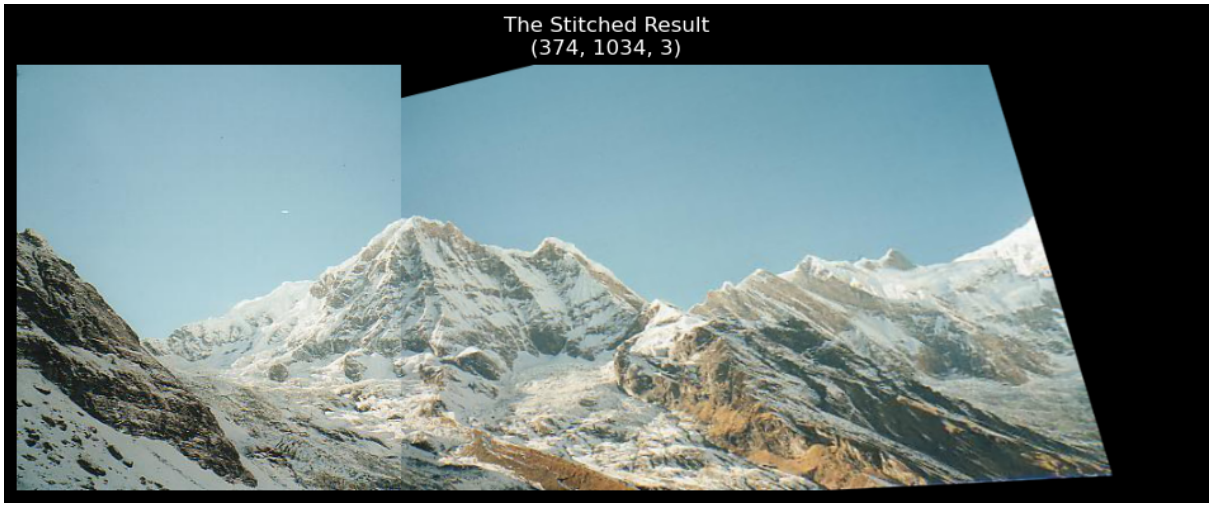

Figure 54 The stitch result from manually selected correspondences

Visually, the new stitch result is more accurate, especially around the stitch line.

Theoretical analysis:

- Biased model assumption: The homography estimation tasks normally assume a planar scene and a pinhole camera model. If the scene is not strictly planar, the estimated homography may introduce bias, especially in regions where the scene deviates from planarity.
- Improper optimisation method: Some homography estimation techniques rely on optimisation methods like RANSAC or least squares. Some methods may converge to local minima or be sensitive to outliers, which result in biased estimates if not appropriately handled.
- Parameterisation: The choice of parameterisation for representing the homography matrix can introduce the estimation bias, such as under-parameterisation or over-parameterisation.
- The distribution of correspondence: The distribution of correspondences significantly influences homography estimation. A uniform distribution allows for accurate estimation across the entire scene, while clustering, outliers, sparse distributions, and non-uniform scales can introduce biases and affect the reliability of the estimated homography.

Empirically analysis:

- Noise in correspondences: Homography estimation methods are sensitive to noise in correspondences between image points. When there are outliers in the correspondences, the estimated homography tends to be biased towards the outliers.
- Insufficient correspondences: Normally, a homography requires a minimum number of correspondences. Insufficient correspondence may be biased due to the limited information available.

- Lens distortion: Homography estimation typically neglects lens distortions. In real-world scenarios, lens distortions can introduce bias, especially in wide-angle lenses or fisheye cameras.
- Scale ambiguity: Homography estimation alone cannot determine scale, leading to scale ambiguity.
- Inaccurate camera calibration: Homography estimation relies on accurate camera calibration parameters. Errors in camera calibration, such as inaccuracies in focal length or principal point estimation, can propagate into biased homography results.