HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Computer Vision Capstone Project

Professor: **PhD**. **Nguyen Thi Oanh**

Students:

1. **Trinh Bao Ngoc - 2018429**
2. **Nguyen Huy Hoang - 20184264**
3. **Le Ngoc Kien - 20184280**
4. **Hoang Minh Tri - 20184315**
5. **Nguyen Tuan Minh - 20184296**

Academic Year 2022 - 2023

Tables of contents

Project 1. Detect Chessboard size. 3

Abstraction 3

1. Introduction 4

2. Related work 4

2.1. Sharpen the image. 4

2.1.1. Frequency class domain 4

2.1.2. Gaussian Blur 6

2.1.3. Threshold 6

2.1.4. Transpose image 7

3. Experimental Result 8

References 10

Contribution 11

Project 2: Object detection using local features. 12

Abstract 12

1. Introduction 13

2. Methodology 15

2.1. Object Detection 15

2.2. SIFT 15

2.3. FLANN 18

3. Demo 20

4. Parameter evaluation. 23

References 25

Contribution 26

Project 1. Detect Chessboard size.

Abstraction

Chessboard corner detection is a necessary procedure of the popular chessboard pattern-based camera calibration technique. The inner corners of a two-dimensional chessboard are employed as calibration markers. In this study, an automatic chessboard corner detection algorithm is presented for camera calibration. A simple yet effective approach is adopted to sort the detected corners into a meaningful order. Finally, the sub-pixel location of each corner is calculated. The proposed algorithm only requires user input of the chessboard size, while all the other parameters can be adaptively calculated with a statistical approach. The experimental results demonstrate that the proposed method has advantages over the popular OpenCV chessboard corner detection method in terms of detection accuracy and computational efficiency.

1. Introduction

Camera calibration is an important technique that has been widely used in many machine vision applications such as stereo vision, visual surveillance, and intelligent transportation. In the last few decades, various camera calibration methods have been introduced, which can be generally grouped into three categories: self-calibration, two-dimensional (2D) planar pattern-based calibration, and 3D object pattern-based calibration. Among them, the calibration methods using 2D planar patterns like circles, grid squares, and concentric circles are very popular for their practical convenience. In particular, the planar chessboard with black-and-white squares is one of the most commonly used patterns for its flexibility. All the chessboard pattern-based calibration methods are on the premise of the accurate detection of chessboard inner corners, which are also known as X-corners for their intuitive perception. Datta et al. verified that the accurate detection of these control points is of great importance to the camera calibration. However, compared with the great concentration on developing robust algorithms for the calculation of camera parameters, less attention has been paid to the bottleneck of X-corner detection.

Currently, there are two widely used camera calibration interfaces.
The first one is a MATLAB toolbox developed by Bouguet. This toolbox is very effective for it has many powerful functions. However, its X-corner detection step asks users to click on the four extreme corners of the chessboard for each calibration image, which limits its practicability to a large extent. The second one is the camera calibration module included in the OpenCV library.

In this module, a chessboard corner detection function named findChessboardCorners is offered. Unlike the manual approach adopted in the MATLAB toolbox, this OpenCV function only requires users to input the size of the chessboard (the number of X-corners in 2D). Generally, the OpenCV method can achieve satisfactory results even in a scene with complex backgrounds. However, it usually fails in working when the area of the square is too small, or the projection angle is too large. Meanwhile, the computational efficiency of this method will significantly decrease when the number of X-corners increases. In this project, the OpenCV method is applied because of its simplification and effectiveness.

2. Related work

2.1. Sharpen the image.

2.1.1. Frequency class domain

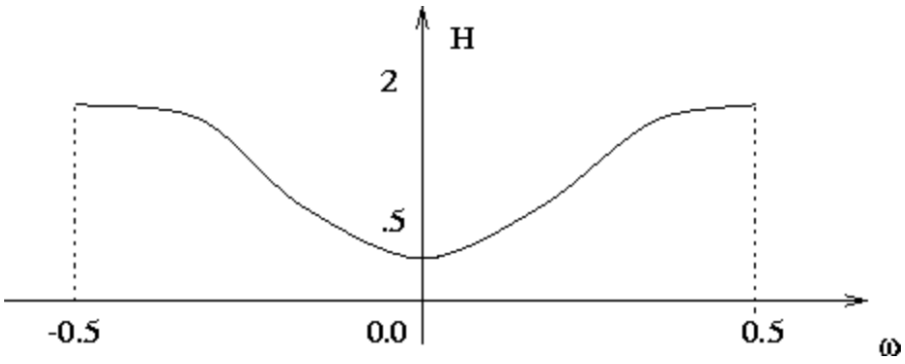
Image enhancement in the frequency domain [3] is straightforward. We simply compute the Fourier transform of the image to be enhanced, multiply the result by a filter (rather than convolve in the spatial domain), and take the inverse transform to produce the enhanced image.

The idea of blurring an image by reducing its high-frequency components or sharpening an image by increasing the magnitude of its high-frequency components is

intuitively easy to understand. However, computationally, it is often more efficient to implement these operations as convolutions by small spatial filters in the spatial domain. Understanding frequency domain concepts is important and leads to enhancement techniques that might not have been thought of by restricting attention to the spatial domain.

Images normally consist of light reflected from objects. The basic nature of the image F(x,y) may be characterized by two components: (1) the amount of source light incident on the scene being viewed, and (2) the amount of light reflected by the objects in the scene. These portions of light are called the illumination and reflectance components and are denoted i(x,y) and r(x,y) respectively. The functions i and r combine multiplicatively to give the image function F:

$$F(x,y) = i(x,y)r(x,y)$$



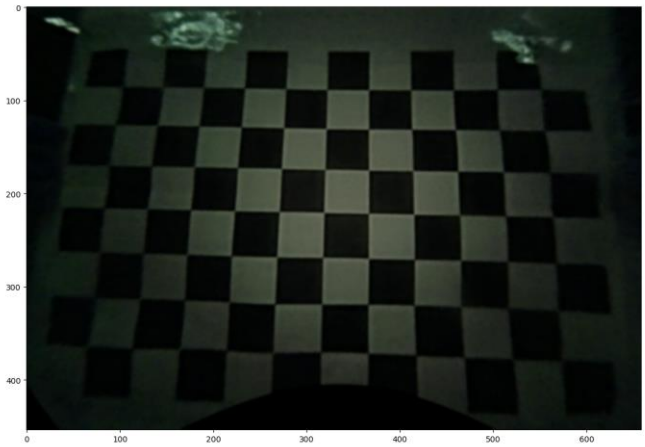
Original **After filtering**

2.1.2. Gaussian Blur

Blurring is to make something less clear or distinct. This could be interpreted quite broadly in the context of image analysis - anything that reduces or distorts the detail of an image might apply. Applying a low pass filter, which removes detail occurring at high spatial frequencies, is perceived as a blurring effect.

Both grayscale and color images can contain a lot of noise, or random variation in brightness or hue among pixels. The pixels in these images have a high standard deviation, which just means there's a lot of variation within groups of pixels. Because a photograph is two-dimensional, Gaussian blur uses two mathematical functions (one for the x-axis and one for the y) to create a third function, also known as a convolution.

This third function creates a normal distribution of those pixel values, smoothing out some of the randomness. How much smoothing depends on the size of the blur radius you choose. Each pixel will pick up a new value set to a weighted average of its surrounding pixels, with more weight given to the closer ones than to those farther away. The result of all this math is that the image is hazier.



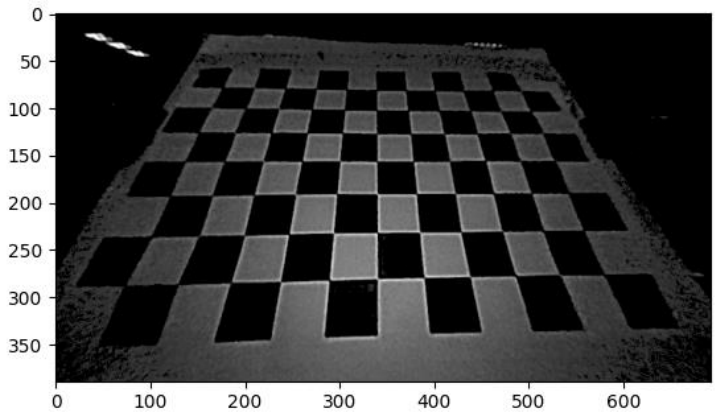
Original **Blurred**

2.1.3. Threshold

Thresholding is a technique in OpenCV, which is the assignment of pixel values in relation to the threshold value provided. In thresholding, each pixel value is compared with the threshold value. If the pixel value is smaller than the threshold, it is set to 0, otherwise, it is set to a maximum value (generally 255). Thresholding is a very popular segmentation technique, used for separating an object considered as a foreground from its background. A threshold is a value which has two regions on its either side i.e. below the threshold or above the threshold.

In Computer Vision, this technique of thresholding is done on grayscale images. So initially, the image must be converted in grayscale color space.

All the pixels in the original image have a grayscale value equal to the value of a number from the pure black (0) to the pure white (255). So, with the thresholding technique we can remove the background color, the image is left with only black and white pixels.



Original **Blurred**

2.1.4. Transpose image

Rotation by an angle θ can be defined by constructing a matrix, *M*, in the form:

$$M = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Given an *(x, y)*-Cartesian plane, this matrix can be used to rotate a vector θ degrees (counterclockwise) about the origin. In this case, the origin is normally the *center* of the image; however, in practice, we can define any arbitrary *(x, y)*-coordinate as our rotation center.

From the original image, *I*, the rotated image, *R*, is then obtained by simple matrix multiplication: $R = I \times M$

However, OpenCV also provides the ability to (1) scale (i.e., resize) an image and (2) provide an arbitrary rotation center around which to perform the rotation.

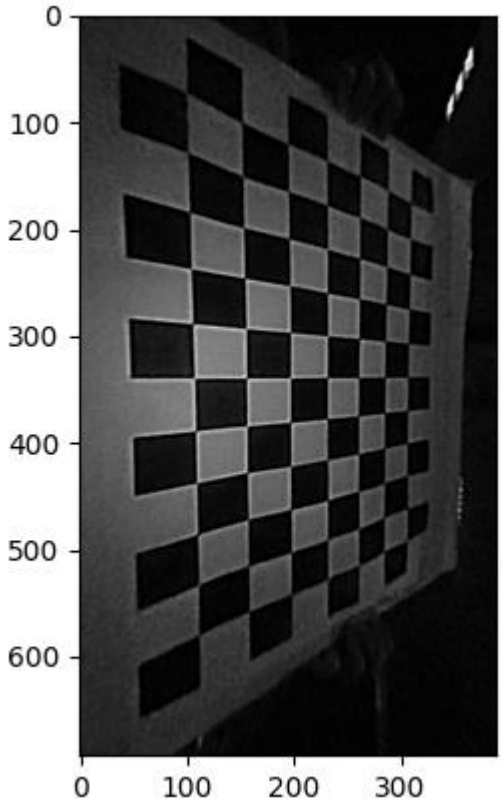
Our modified rotation matrix, M, is thus:

$$M = \begin{bmatrix} \alpha & \beta & (1-\alpha) \times c_x - \beta \times c_y \\ -\beta & \alpha & \beta \times c_x + (1-\alpha) \times c_y \end{bmatrix}$$

where $\alpha = \text{scale} \times \cos\theta$ and $\beta = \text{scale} \times \sin\theta$ and c_x and c_y are the respective *(x, y)*-coordinates around which the rotation is performed.

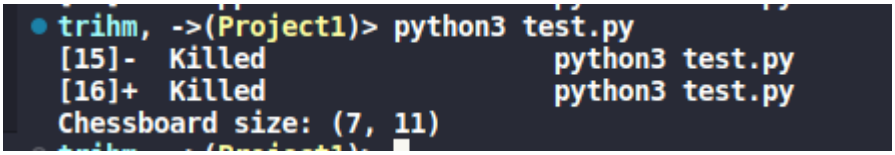
Since the chessboard in the original images is placed at a different angle, it causes a lot of problems in determining the size. To solve, we have to make the

chessboard stand up as much as possible, so we choose to transpose the image, rotate it by 90 degrees.

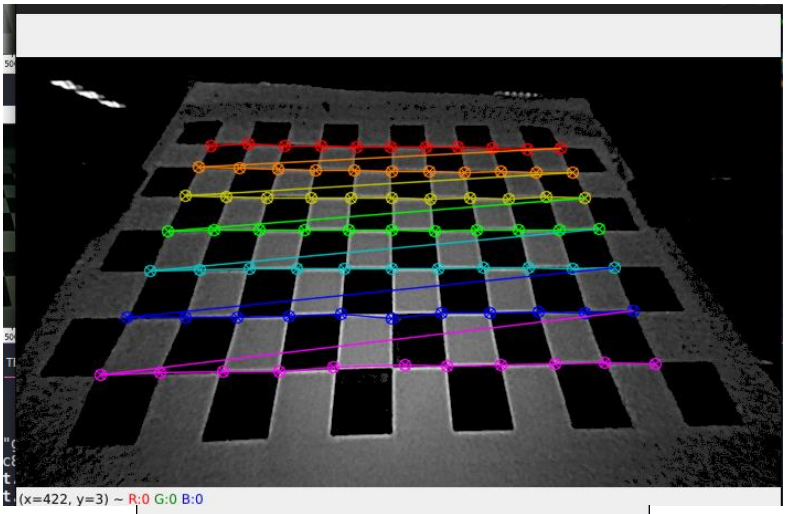


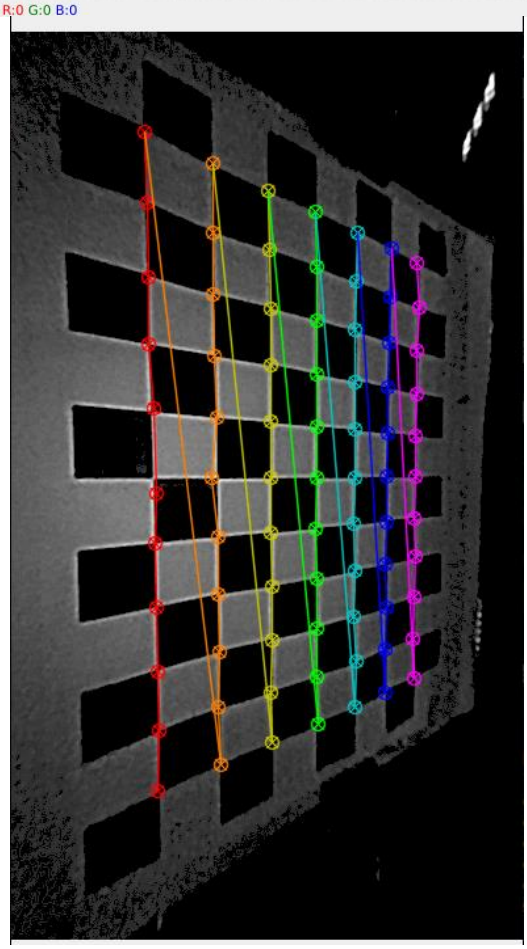
3. Experimental Result

Since we apply the method that find all the inner corners of the chessboard, just take the result + 1 (border) and we get the exact result. For example: a chessboard size with (12, 8) will have the (11, 7) inner corners



Here is the result of drawing all the found corners: (Some images are still able to detect the corner without being transposed)


(x=422, y=3) ~ R:0 G:0 B:0



References

[1]https://www.researchgate.net/publication/282446068_Automatic_chessboard_corner_detection_method
[2]https://docs.opencv.org/4.x/d4/d13/tutorial_py_filtering.html
[3]https://en.wikipedia.org/wiki/Frequency_domain
[4]https://www.geeksforgeeks.org/python-image-blurring-using-opencv/
[5]https://stackabuse.com/opencv-thresholding-in-python-with-cv2threshold/
[6]https://www.geeksforgeeks.org/python-thresholding-techniques-using-opencv-set-1-simple-thresholding/
[7]https://www.geeksforgeeks.org/python-opencv-cv2-transpose-method/
[8]https://docs.opencv.org/4.x/d4/d13/tutorial_py_filtering.html
[9]https://en.wikipedia.org/wiki/Gaussian_blur

Contribution

Name	Work
Hoàng Minh Trí (50%)	Report, Code, Slides
Nguyễn Tuấn Minh (50%)	Report, Code, Slides

Project 2: Object detection using local features.

Abstract

Object detection using local features is a computer vision technique that uses local features such as color, texture, and shape to detect objects in an image. It is used in a variety of applications such as object tracking, facial recognition, pedestrian detection, and scene segmentation. The goal of object detection is to accurately detect objects in an image, and then assign a label or class to each detected object. Local features are typically used to describe objects in an image. Examples of local features include color, texture, shape, and edge information. For example, a color histogram could be used to detect a particular color in an image, or a texture descriptor could be used to identify a particular texture in an image. Local features can be used to provide a more precise description of an object than a single global feature. This enables object detection systems to be more accurate and robust.

1. Introduction

This capstone project presents an in-depth exploration of the use of local features for object detection. The report begins with an overview of the current state of the art in object detection, followed by a focused review of the various local feature detection algorithms available. Following this, an empirical comparison of the most popular local feature detection methods is made, based on their performance on a standard dataset of images. Finally, the report concludes with an analysis of the results and an outline of potential directions for further research.

Object detection is a computer vision technique that allows us to identify and locate objects in an image or video. With this kind of identification and localization, object detection can be used to count objects in a scene and determine and track their precise locations, all while accurately labeling them.

Let's simplify these statements a bit with the help of below images:

Figure 1. Template images of dogs

So instead of classifying which types of dogs are present in these images, we must locate the position of the dog in the image if it exists. When detecting the position of the template images of the dog, we can create a box around the dog with specific the x and y coordinates of this box.

Figure 2. The images with detected box

Object detection is the task of finding and classifying objects in an image. It is a very important task in computer vision and has a wide range of applications. One of the most popular approaches to object detection is using local features. Local features are features of an object that are unique to that object, and are invariant to transformations such as scaling, rotation, and illumination. Examples of local features include corners, edges, and blobs. Object detection using local features typically involves a two-step process. First, the local features in the image are extracted. Then, these features are used to find objects of interest in the image.

To extract local features from an image, methods such as Scale-Invariant Feature Transform (SIFT) and Speeded Up Robust Features (SURF) are often used. These methods find local features in the image and generate a set of feature descriptors that are invariant to transformations. Once the local features have been extracted, they can then be used to detect objects in the image. This is usually done by comparing the feature descriptors of the image to a set of feature descriptors of known objects. If a match is found, then the object can be detected.

Object detection using local features is a powerful technique that has many applications, such as facial recognition, object tracking, and image classification. This method is popular because it is fast and accurate. It is also robust to transformations such as scaling and rotation. Object detection using local features is a powerful technique that has many applications. This method is popular because it is fast and accurate and is robust to transformations.

2. Methodology

2.1. Object Detection

a. Introduction

Object detection is a computer technology related to computer vision and image processing that deals with detecting instances of semantic objects of a certain class (such as humans, buildings, or cars) in digital images and videos. Numerous computer vision technologies, such as image retrieval and video surveillance, utilize object detection. Basically, object detection is a modern computer technology that is related to image processing, deep learning, and computer vision to detect the objects present in an image file. All the technologies used in the Object detection technique (as we mentioned earlier) deals with detecting instances of the object in the image or video.

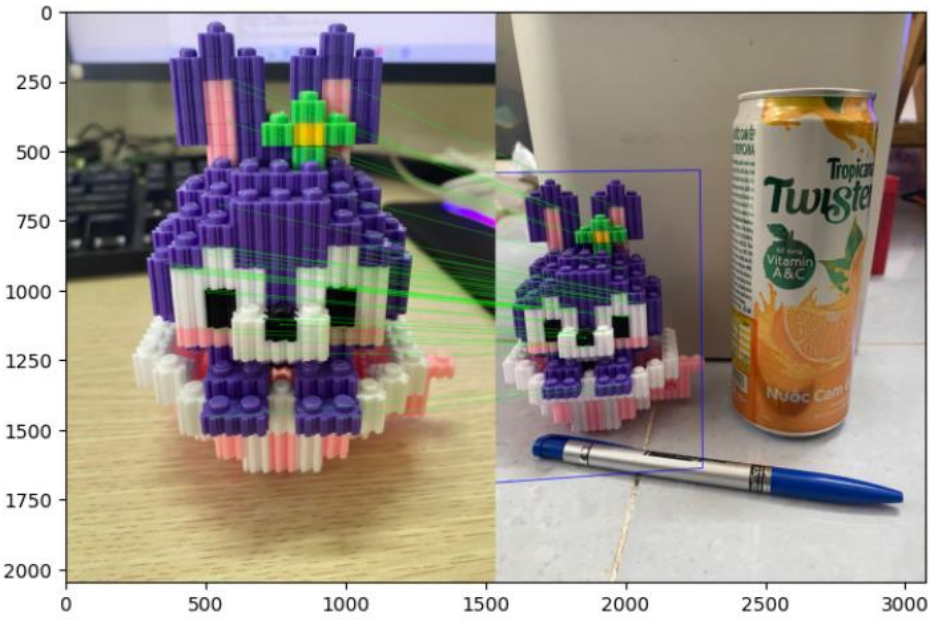


Fig 3. An example of object detection

2.2. SIFT

Scale-Invariant Feature Transform (**SIFT**) is an algorithm in computer vision to detect and describe local features in images. It is a feature that is widely used in image processing. The processes of SIFT include Difference of Gaussians (DoG) Space Generation, Keypoints Detection, and Feature Description.

It is used in a variety of applications, including object recognition, robotic mapping and navigation, image stitching, 3d modeling, gesture recognition, video tracking, individual identification of wildlife and match moving.

The steps for SIFT are as follows:

- **Scale-space Extrema Detection**
- **Keypoint Localization**
- **Orientation Assignment**
- **Keypoint Descriptor**

Scale-space extrema detection in SIFT is a process in which local maxima and minima of the difference-of-Gaussian (DoG) function are located. By comparing each pixel with its nine neighbors in the next lower and higher scale in the scale-space, as well as its eight nearby neighbors, these extrema are discovered. By comparing these extrema to the maxima and minima in their immediate vicinity, these extrema are then further refined. This aids in locating the most consistent and recognizable elements.

Key point localization follows. Numerous key points are produced from the key points established in the previous stage. Some of them are too close to the edge or lack contrast. They are not as helpful as features in either situation. So, we eliminate them. The method for deleting edge characteristics is comparable to that employed in the Harris Corner Detector. We merely check the intensities of low-contrast features.

For a more precise placement of the extrema, they expanded the scale space using the Taylor series, and if the intensity at the extrema is less than a certain threshold (according to the paper, 0.03), it is rejected. Edges must also be removed because DoG responds more strongly to them. They used a 2x2 Hessian matrix (H) to compute the principal curvature.

In the **orientation assignment**, a neighborhood is taken around the key point location depending on the scale, and the gradient magnitude and direction is calculated in that region. An orientation histogram with 36 bins covering 360 degrees is created. And the "amount" that is added to the bin is proportional to the magnitude of the gradient at that point. Once you've done this for all pixels around the key point, the histogram will have a peak at some point. The highest peak in the histogram is taken and any peak above 80% of it is also considered to calculate the orientation. It creates key points with the same location and scale, but different directions. It contributes to the stability of matching.

At this point, each key point has a location, scale, orientation. Next is to **compute a descriptor** for the local image region about each key point that is highly distinctive and invariant as possible to variations such as changes in viewpoint and illumination.

To do this, a 16x16 window around the key point is taken. It is divided into 16 sub-blocks of 4x4 size. For each sub-block, an 8-bin orientation histogram is created. So, 4 X 4 descriptors over 16 X 16 sample arrays were used in practice. 4 X 4 X 8 directions give 128 bin values. It is represented as a feature vector to form a key point descriptor.

a. Parameters:

SIFT (**Scale -Invariant Feature Transform**) has 4 important parameters:

- **Number of Octaves:** The number of octaves used in the feature detection. This determines the number of different scales at which the feature points must be detected. A higher number of octaves allows for the detection of features over a greater range of scales; however this comes at the cost of increased computation time.
- **Number of Octave Layers:** The number of octave layers used in the feature detection. This determines the level of detail that is captured in the feature descriptor. More layers may provide more features to identify, leading to more accurate detection. However, this could lead to a decrease in performance and increase the computational cost of the algorithm if too many layers are used.

- **Edge Threshold:** The edge threshold is used to filter out weak edges. It determines the strength of an edge to determine whether or not it should be included in the feature descriptor. With a higher edge threshold, the algorithm will detect fewer features, as it only detects features that are of sufficient contrast. This can lead to fewer features being detected, making the matching process more difficult. Conversely, a lower edge threshold can lead to more features being detected, making the matching process easier. Additionally, a lower edge threshold may lead to more false positives, meaning that the algorithm will detect features which are not actually present in the image.
- **Contrast Threshold:** The contrast threshold is used to filter out weak features. It determines the amount of contrast between adjacent pixels to determine whether a feature should be included in the feature descriptor. Increasing the contrast threshold will result in fewer detected points, while decreasing the contrast threshold will result in more detected points. This can result in a different set of features being detected, and thus a different outcome for the SIFT algorithm.

b. Pros and Cons:

Pros of SIFT

- Able to recognize objects in images and videos regardless of their size, orientation, and illumination.
- Provides high-level understanding of the image content.
- Allows for the accurate matching of local features across multiple images.

Cons of SIFT

- Expensive in terms of computation power and memory.

The best use cases for S IFT are object recognition, robotic mapping and navigation, image stitching, 3D modeling, gesture recognition, video tracking, and wildlife identification. The worst cases of SIFT are those that require large amounts of computational power, or when the feature size is too small or noisy for accurate detection.

c. Mode of Operation

First, we need to construct a SIFT object using

```
sift = cv.SIFT_create()
```

We can pass different parameters to it, here we are just using the parameters by default. We also need to convert the image to grayscale:

```
image_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Detect and compute the descriptors of keypoints using

```
key points, descriptors = sift.detectAndCompute(image_gray,None).
```

Apply the process for both the templates and images to detect and describe the local features of them. We can implement a method to draw the keypoints on the images for better visualization.

```
def draw_keypoints_sift(keypoints,image,image_gray):
    final_image=cv2.drawKeypoints(image_gray,keypoints,image,flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    plt.imshow(final_image)
```

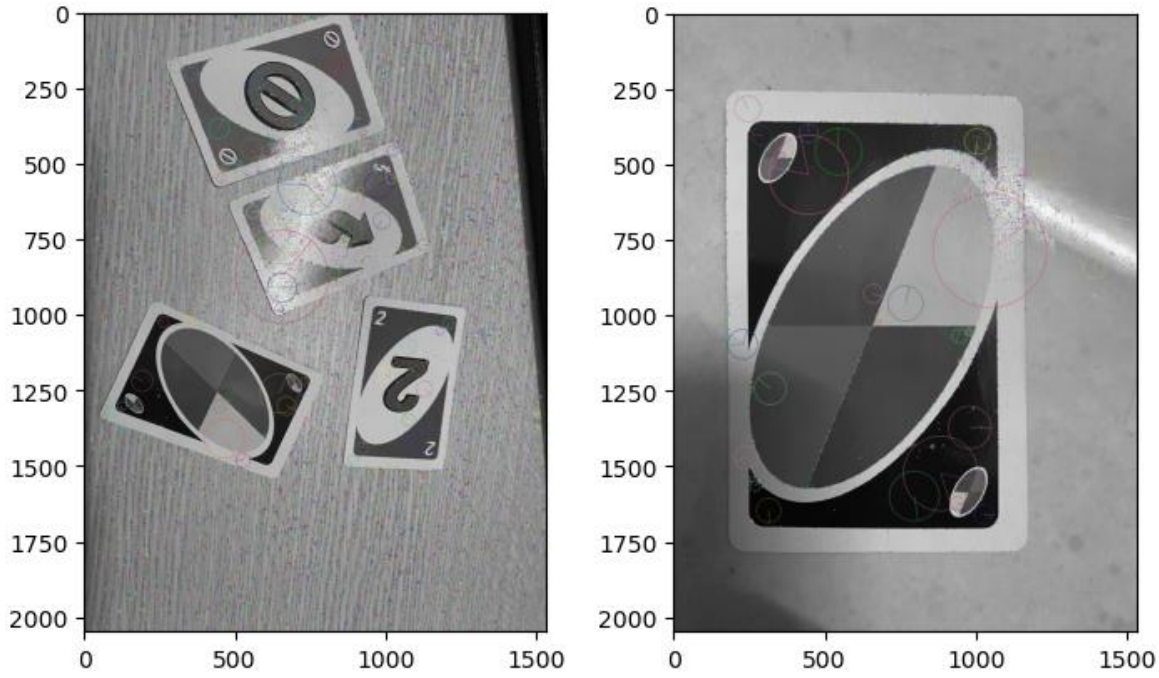


Fig 4&5. FLANN image matching with detected object

2.3. FLANN

a. Introduction

FLANN, an acronym for **Fast Library for Approximate Nearest Neighbors**, is a C++ library for performing fast approximate nearest neighbor searches in high dimensional spaces. It contains a collection of algorithms we found to work best for nearest neighbor search and a system for automatically choosing the best algorithm and optimum parameters depending on the dataset. **FLANN** is written in C++ and contains bindings for the following languages: C, MATLAB, Python, and Ruby.

b. Parameters

Flann-based descriptor matcher.
This matcher trains cv::flann::Index on a train descriptor collection and calls its nearest search methods to find the best matches. So, this matcher may be faster when matching a large train collection than the brute force matcher. FlannBasedMatcher does not support masking permissible matches of descriptor sets because flann::Index does not support this. There are 2 main param in the FlannBasedMatcher:

- First one is **IndexParams**. For various algorithms, the information to be passed is explained in FLANN docs. As a summary, for algorithms like SIFT, SURF etc. you can pass following:
  index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
  We use **SIFT** for the implementation but for using **ORB** (An efficient alternative to **SIFT** or **SURF**), you can pass the following. The commented values are recommended as per the docs, but it didn't provide required results in some cases. Other values worked fine:
  index_params= dict(algorithm = FLANN_INDEX_LSH, table_number = 6,
  key_size = 12, multi_probe_level = 1)

- Second is the **SearchParams**. It specifies the number of times the trees in the index should be recursively traversed. Higher values gives better precision, but also takes more time. If you want to change the value, pass:
  search_params = dict(checks=50)

c. Mode of Operation

After detecting and computing the key points and descriptors in the both input images using sift.detectAndCompute(), we create a FLANN based matcher object flann=cv2.FlannBasedMatcher(index_param,search_params) and match the descriptors using flann.knnMatch(des1,des2,k=2), which returns the matches.

We also keep only stable point pairings (m.distance: nearest distance, n.distance: 2nd nearest distance): m.distance < distance_ratio*n.distance; We determine the minimum number of pairs of points found to be about 10 pairs.

Next, we locate the corresponding object found between source point and destination point, finding the similarity transformation between two matched sets of points:

```
M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC,5.0)
```

Then determine the corresponding positions on the destination image of the 4 corners on the source image and perform the transform to find the corresponding position in the matched image:

```
pts = np.float32([ [0,0],[0,h-1],[w-1,h-1],[w-1,0] ]).reshape(-1,1,2)
dst = cv2.perspectiveTransform(pts,M)
```

Apply the ratio test on the matches to obtain best matches. Draw the matches using

```
draw_params = dict(matchColor = (0,255,0), # draw matches in green color
                   singlePointColor = None,
                   matchesMask = matchesMask, # draw only inliers
                   flags = 2)
cv2.drawMatches(img1, kp1, c_img2, kp2, good, None, **draw_params).
```

And then visualize the keypoint matches, as we show some examples below.

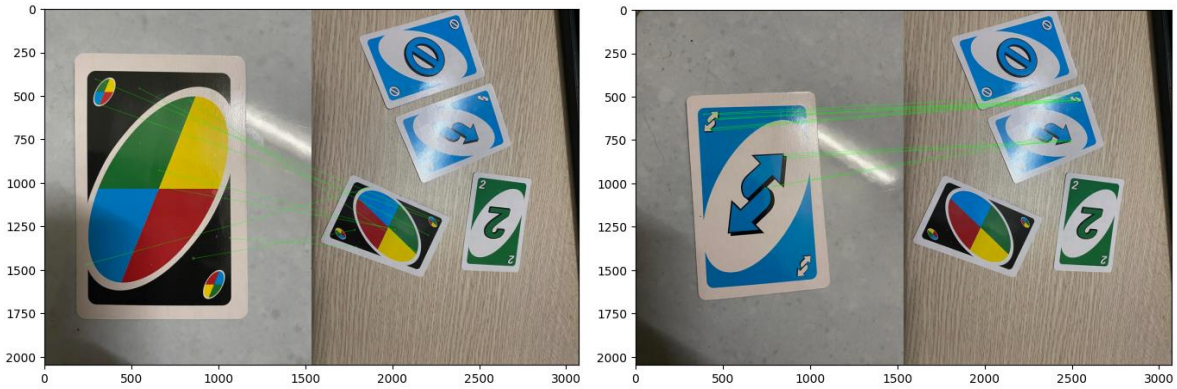


Fig 6&7. FLANN image matching with detected object

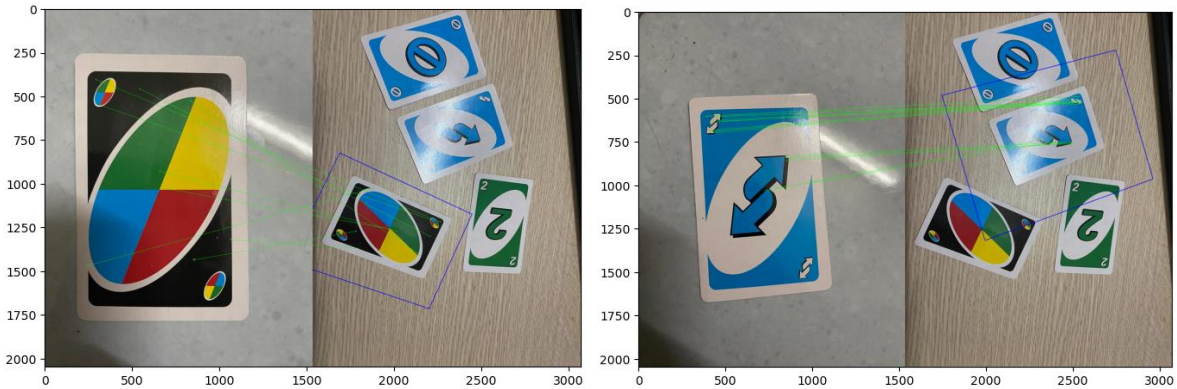


Fig 8&9. FLANN image matching with detected object

d. Pros & cons

These methods project the high-dimensional features to a lower-dimensional space and then generate the compact binary codes. Benefiting from the produced binary codes, fast image search can be carried out via binary pattern matching or Hamming distance measurement, which dramatically reduces the computational cost and further optimizes the efficiency of the search, which means a good result - O(logN). But on the other hand, the implementation is quite long, and finding a threshold is difficult compared to some other ways.

3. Demo

In this demo, we divide the detection process into two parts: local feature extraction and image matching.

- **Local Feature Extraction**

```
## SIFT
def draw_keypoints_sift(keypoints,image,image_gray):
    final_image=cv2.drawKeypoints(image_gray,keypoints,image, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    plt.imshow(final_image)

def sift_local_feature_extraction(img):
    sift = cv2.SIFT_create()
    image_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # find keypoints, descriptor
    keypoints, descriptors = sift.detectAndCompute(image_gray,None)
    #draw_keypoints_sift(keypoints,img,image_gray)
    dic = dict()
    dic['kp'] = keypoints
    dic['des'] = descriptors
    return dic
templates = load_images_from_folder('./templates/uno')
images = load_images_from_folder('./images/uno')
#plt.imshow(cv2.cvtColor(templates[0], cv2.COLOR_BGR2RGB))
template_kp_des = sift_local_feature_extraction(templates[0])
image_kp_des = sift_local_feature_extraction(images[0])
template_kp_des
```

Fig 10. SIFT python code.

In the upper source code, we apply SIFT local features extraction using opencv to implement it. First, we convert the image into gray scale base with **cvtColor** function. Then using the **sift** variable (created by called SIFT_create function from **cv2** library) to call **detectAndCompute** function to extract the keypoints and descriptors of the image.

Each key point is a special structure that has many properties such as its coordinates, the size of its neighborhood, the angle that determines its direction, the feedback that specifies the magnitude of the key points,…

```
Out[8]: {'kp': (< cv2.KeyPoint 000002C97D186600>,
                < cv2.KeyPoint 000002C900D19210>,
                < cv2.KeyPoint 000002C900D192A0>,
                < cv2.KeyPoint 000002C900D192D0>,
                < cv2.KeyPoint 000002C900D19270>,
                < cv2.KeyPoint 000002C900D19240>,
                < cv2.KeyPoint 000002C900D193C0>,
                < cv2.KeyPoint 000002C900D19390>,
                < cv2.KeyPoint 000002C900D19420>,
                < cv2.KeyPoint 000002C900D19450>,
                < cv2.KeyPoint 000002C900D19480>,
                < cv2.KeyPoint 000002C900D194B0>,
                < cv2.KeyPoint 000002C900D194E0>,
                < cv2.KeyPoint 000002C900D19510>,
                < cv2.KeyPoint 000002C900D17420>,
                < cv2.KeyPoint 000002C900D17F90>,
                < cv2.KeyPoint 000002C900D17EA0>,
                < cv2.KeyPoint 000002C900D17E70>,
                < cv2.KeyPoint 000002C900D17CC0>,
```

Fig 11. Result of the detectAndCompute function of cv2.

OpenCV also provide the function **drawKeyPoints** to draw the circle around the position of the keypoint. In our code, we use flag: draw_matches_flags_rich_keypoints to determine the circle with size of the keypoint with its direction.

Fig 12. Image of the keypoints with circle round.

- **Image matching**

In our project, we use FLANN algorithms to match the template of the which we want to find in the target image.

First, we must determine the local features of two images: template and target image. By using SIFT function mentioned above, we have two dictionaries of the keypoints and descriptors. Then with the FLANN in the cv2 library, we can implement the image matching.

- Create the flann object by calling the **FlannBaseMatcher** function.
- To find the maches between two images, **knnMatch** function of the flann object help us to do that.
- Use the Lowe's ratio test to find the best matches of these matching lines.
- Visualize the matching lines with **drawMatches** function of the cv2 library.

```python
# FLANN
def flann_image_matching(img1,img2,kp1,kp2,des1,des2,distance_ratio = 0.7):
    FLANN_INDEX_KDTREE = 0
    index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
    search_params = dict(checks=50)

    c_img2 = img2.copy()

    flann = cv2.FlannBasedMatcher(index_params,search_params)

    matches = flann.knnMatch(des1,des2,k=2) # trả về 2 features gần nhất từ tập
des2
    matchesMask = [[0,0] for i in range(len(matches))]
    MIN_MATCH_COUNT=10
    good = []
    for m,n in matches:
        if m.distance < distance_ratio*n.distance:
            # chỉ giữ lại những ghép cặp ổn định (m.distance: khoảng cách gần
nhất, n.distance: khoảng cách gần thứ 2)
            good.append(m)
    ## khoanh vùng đối tượng tương ứng được tìm thấy
    if len(good) > MIN_MATCH_COUNT:
        src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1,1,2)
        dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1,1,2)

        # tìm phép biến đổi đồng dạng giữa 2 tập điểm được so khớp
        M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC,5.0)

        matchesMask = mask.ravel().tolist()
        h,w,d = img1.shape

        # xác định vị trí tương ứng trên ảnh đích của 4 góc trên ảnh nguồn

        pts = np.float32([ [0,0],[0,h-1],[w-1,h-1],[w-1,0] ]).reshape(-1,1,2) # 4
điểm khoanh vùng trên ảnh nguồn
        dst = cv2.perspectiveTransform(pts,M) # thực hiện biến đổi để tìm ra vị
trí tương ứng trong ảnh được so khớp

        # hiển thị kết quả tương ứng
        c_img2 = cv2.polylines(c_img2,[np.int32(dst)],True,255,2, cv2.LINE_AA)
    else:
        print( "Not enough matches are found - {}/{}".format(len(good),
MIN_MATCH_COUNT) )
        matchesMask = None
    ## hiển thị kết quả
```

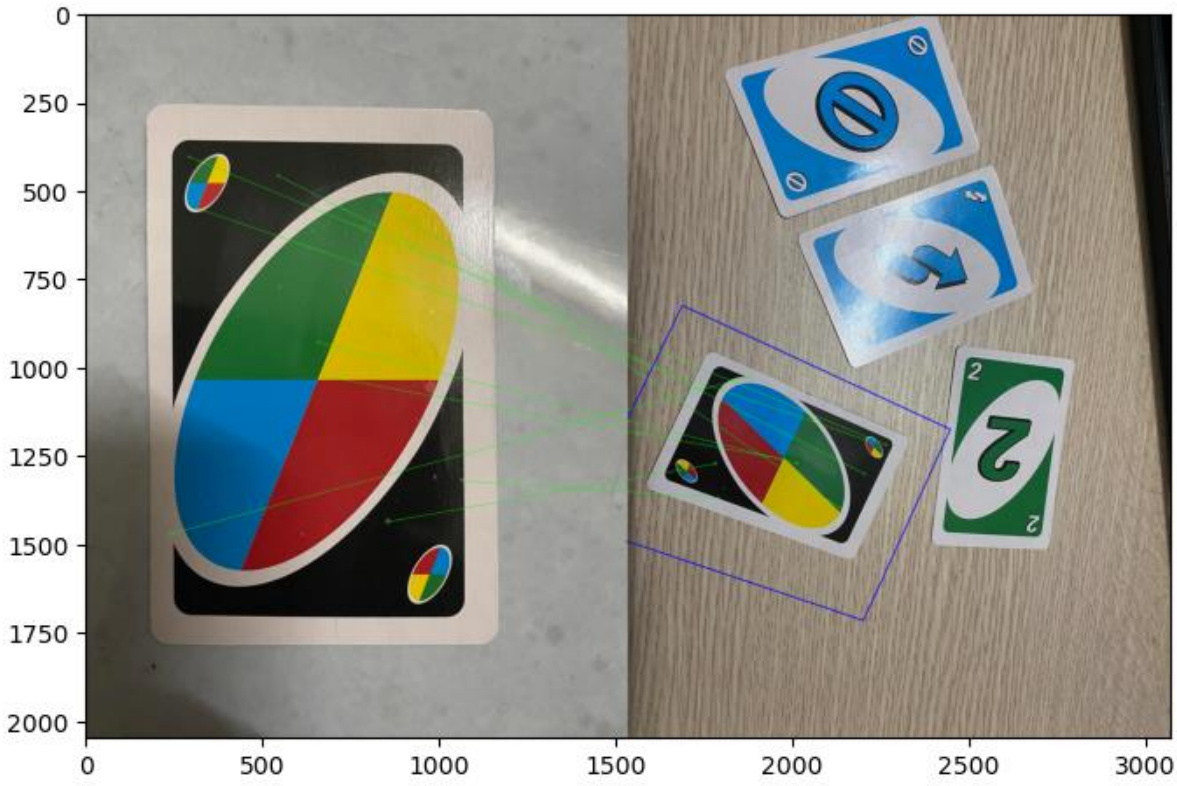


Fig 13. The result when running FLANN matcher (distance_ratio=0.7).

4. Parameter evaluation.

FLANN

- **Distance_ratio:**

Each keypoint of the first image is matched with several keypoints from the second image. We keep the two best matches for each keypoint. Lowe's test checks that two distances are sufficiently different. If they are not, the keypoint is eliminated and will not be used for further calculations.

In detail, first the algorithm compute the distance between feature f_i in the image 1 and all the features f_j in the image 2. Next, it chooses feature f_c in the image 2 with the minimum distance to feature f_i in the image of one as our closest match. And then proceed to get the f_s the feature in the image two with the second closest distance to the feature f_i. Continue finding how much nearer our closest match f_c is over our second closest match f_s through the distance ratio. Finally, we keep the matches with distance ratio < distance_ratio threshold.

Object Detection

In [57]:
```
# run with: toy, thaprua, uno
templates = load_images_from_folder('./templates/thaprua')
images = load_images_from_folder('./images/thaprua')

# Implementation
def object_detection(templates,images):
    for image in images:
        for template in templates:
            template_kp_des = sift_local_feature_extraction(template)
            image_kp_des = sift_local_feature_extraction(image)
            flann_image_matching(template,image,template_kp_des['kp'],image_kp_des['kp'],
                                 template_kp_des['des'],image_kp_des['des'],0.01)

object_detection(templates,images)
```

```
Not enough matches are found - 3/60
```

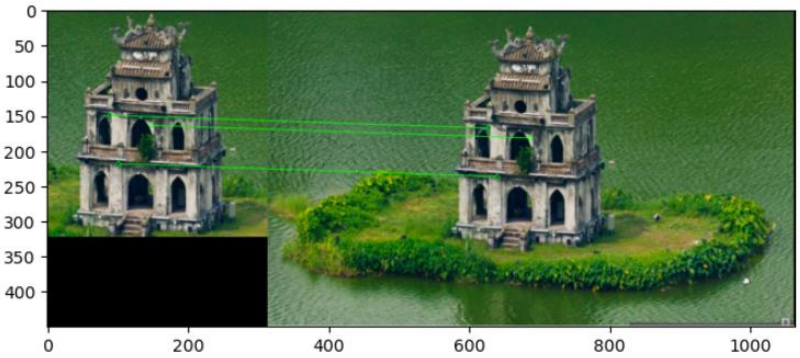


Fig 14. The best matching lines with distance(0.01)

Object Detection

In [65]:
```
# run with: toy, thaprua, uno
templates = load_images_from_folder('./templates/thaprua')
images = load_images_from_folder('./images/thaprua')

# Implementation
def object_detection(templates,images):
    for image in images:
        for template in templates:
            template_kp_des = sift_local_feature_extraction(template)
            image_kp_des = sift_local_feature_extraction(image)
            flann_image_matching(template,image,template_kp_des['kp'],image_kp_des['kp'],
                                 template_kp_des['des'],image_kp_des['des'],0.1)

object_detection(templates,images)
```

```
Matches are found - 630
```

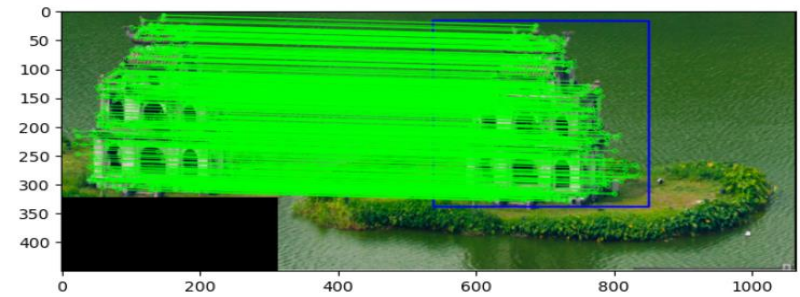


Fig 15. The best matching lines with distance(0.1)

- **MIN_MATCH_COUNT:**

The min_match_count value to determine the lower bound to guarantee the object can be right dectected.

References

1. https://www.analyticsvidhya.com/blog/2022/03/a-basic-introduction-to-object-detection/
2. OpenCV: Introduction to SIFT (Scale-Invariant Feature Transform)
3. OpenCV: cv::SIFT Class Reference
4. OpenCV: Feature Matching with FLANN
5. Feature Matching — OpenCV-Python Tutorials beta documentation (opencv24-python-tutorials.readthedocs.io)
6. OpenCV: cv::FlannBasedMatcher Class Reference
7. https://docs.opencv.org/3.4/d1/de0/tutorial_py_feature_homography.html

Contribution

Name	Work
Trịnh Bảo Ngọc (33%)	Code, Report
Nguyễn Huy Hoàng (33%)	Code,Report,Slides
Lê Ngọc Kiên (33%)	Code,Report,Slides