Homework 02

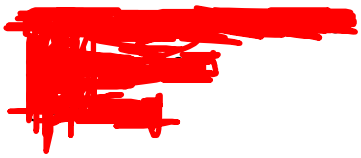


Table of Contents

II. User guide

There are several features that you can execute: To execute the .exe file, you need to run the following command:

```
C:path/to/Release>Lab02.exe <1> <2> <3> <4> <5>
```

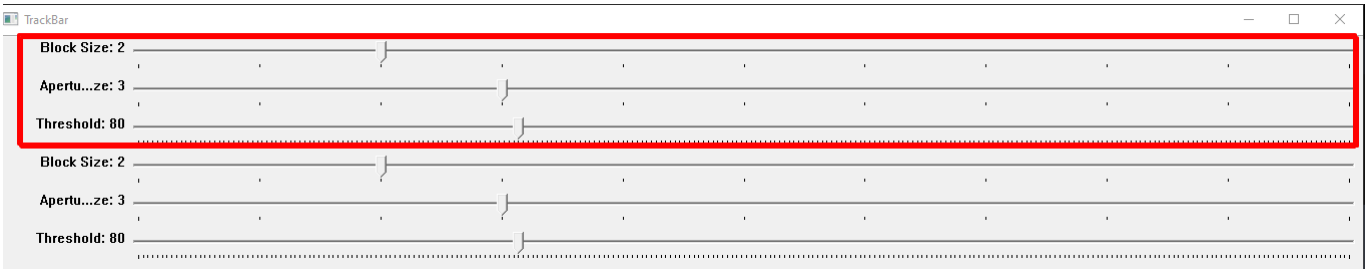
where `0` means that you can not need to write in cmd line

```
<1>
    {
        0, m, h
    }

<2>
    {
        harris, blob, dog
    }

<3>
    {
        0, sift, lbp
```

```
    }

<4> and <5>
    {
        0, path to image1 and image2 respectively
    }
```

You can press `h` while the program is running, it will show helper guideline.

In case processing matching images, the trackbar will show



But you must `focus on` the parameters in the red rectangle and `ignore` the below parameters because they are redundant (occur when calling back multiple time).

My program also supports capturing images from camera. To execute from camera, you just remove `<4>` and `<5>` in command line arguments.

III. Implementations

1. Harris Detection

- Source: Harris.cpp
- Defines a function `HarrisDetection` that applies corner detection on an input image.
- The `cornerHarrisModify` function calculates Harris corners based on trackbar-adjustable parameters such as `block size`, `aperture size`, and `threshold`.

```
void HarrisDetection(Mat inputImage) {
    ...
    harrisParams.blockSize = 2; // max = 10
    harrisParams.apertureSize = 3; // max = 10
    harrisParams.thresh = 150; // max = 255
    double k = 0.04;
    ...
}
```

- Detected corners are highlighted and displayed in real-time on the image by using openCV function such as `cornerHarris`, `normalize`, and `drawKeypoints`.
- Trackbars enable interactive parameter tuning for corner detection.

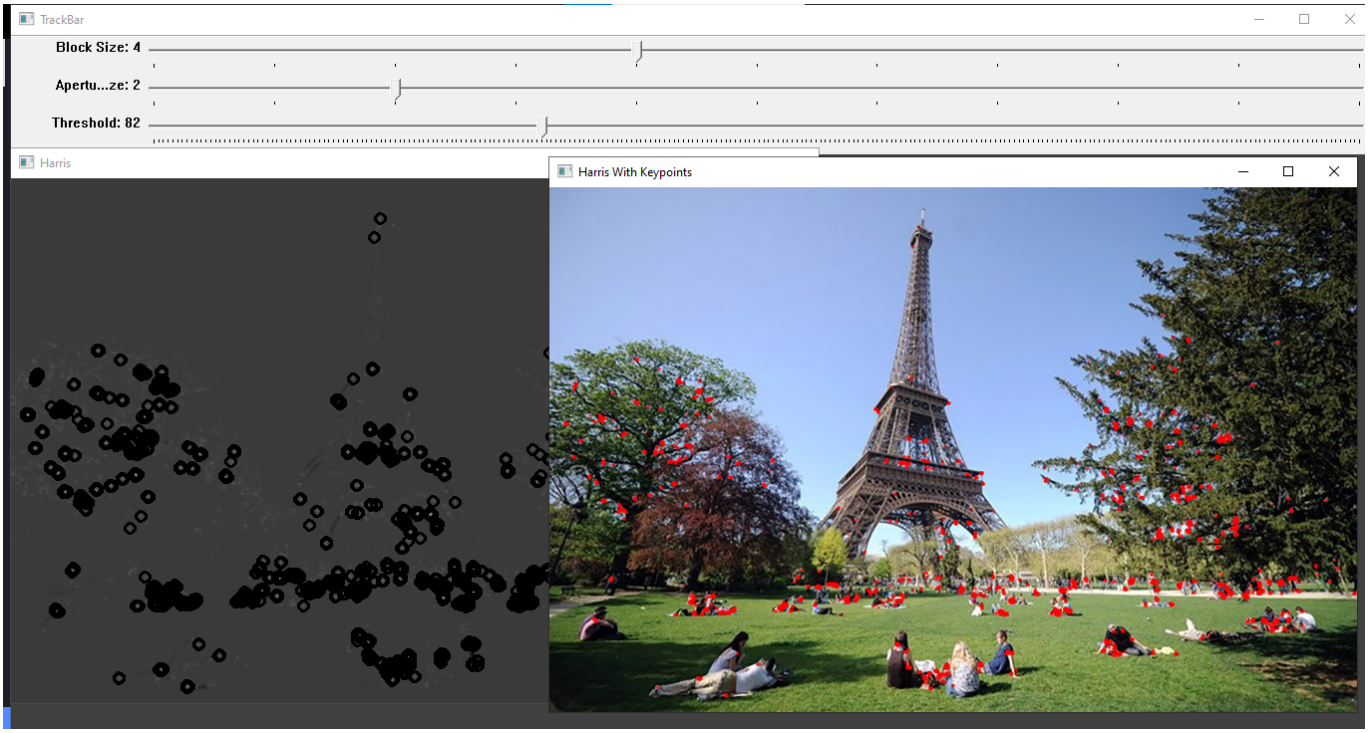


Figure 1: Harris Detector

2. Blob Detection

- Source: Blob.cpp
- Performs Blob detection using OpenCV's `SimpleBlobDetector` function. In this algorithm, I set all parameters with min and max without using `trackbar` to modify.
- Defines a function `BlobDetection` that applies Blob detection on an input image.
- Parameters for `circularity`, `inertia`, `convexity`, `color`, `area`, and `thresholds` are set to filter and identify blobs.

```
void BlobDetection(Mat inputImage) {
    SimpleBlobDetector::Params params;

    params.minCircularity = 0.1;
    params.minInertiaRatio = 0.01;
    params.minConvexity = 0.7;
    params.blobColor = 255;
    params.minThreshold = 10;
    params.maxThreshold = 255;
    params.minArea = 300;
    params.maxArea = 1000;
    ...
}
```

- Detected blobs are highlighted and displayed on the image using OpenCV's `detect` from `SimpleBlobDetector` and `drawKeypoints` functions.

Figure 2: Blob Detector

3. DoG Detection

- Source: DoG.cpp
- Defines a function `DoGDetection` that computes the DoG by applying Gaussian blurs with varying kernel sizes and sigmas to an input image.
- Parameters include `kernel sizes`, `sigmas`, and a `threshold` for feature detection are adjustable through `trackbars` which are applied to compute Gaussian blur.

```
void DoGDetection(Mat inputImage) {
    ...
    dogParams.ksize1 = 3; // max = 21
    dogParams.ksize2 = 7; // max = 21
    dogParams.sigma1 = 1; // max = 10
    dogParams.sigma2 = 2; // max = 10
    dogParams.thresh = 40; // max = 255
    ....
}
```

- Detected keypoints are extracted using OpenCV's `FastFeatureDetector` on the computed DoG result. In first implementation, I used `SIFT` but I'd found the next requirement using SIFT, so I use `ORB` instead.
- The identified keypoints are displayed in real-time on the original image via the `drawKeypoints` function, showcasing the areas of interest based on the DoG computation.

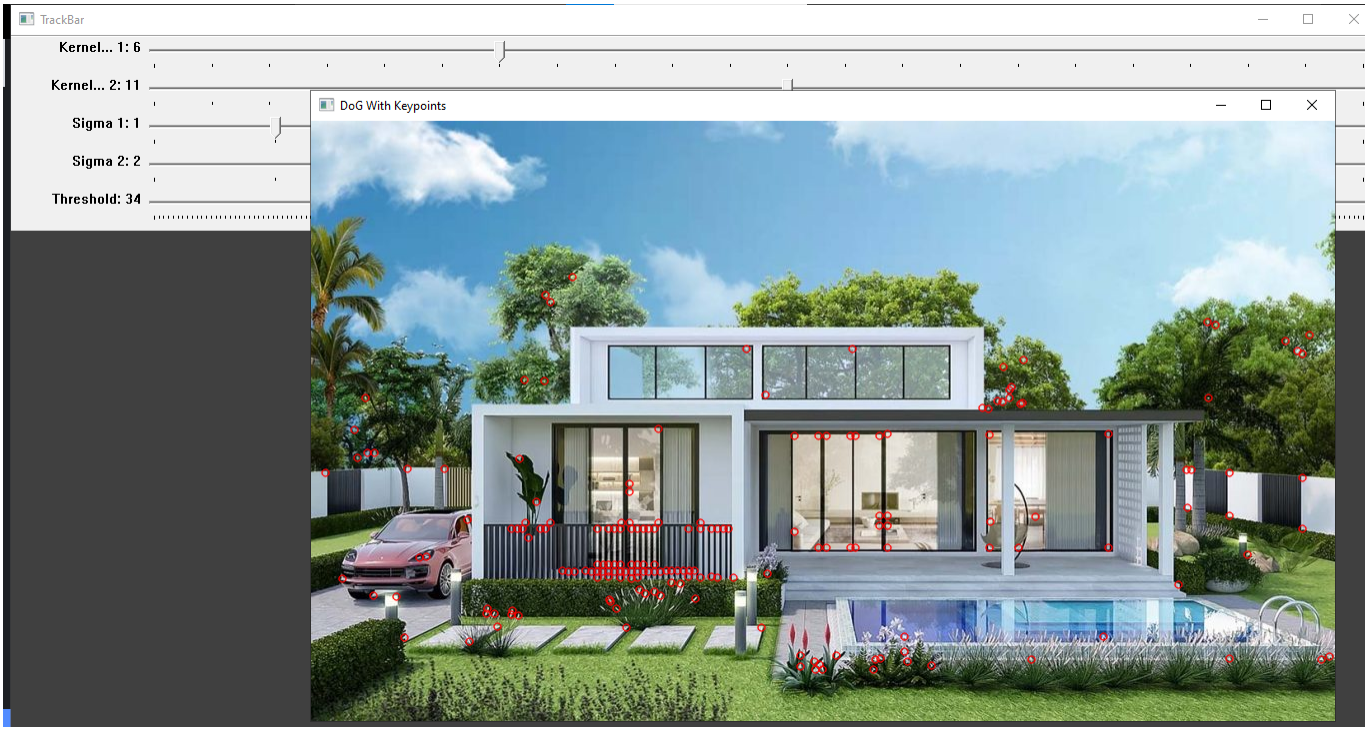


Figure 3: DoG Detector

4. SIFT Descriptor

- Source: SIFT.cpp
- After detecting and extracting keypoints from above 3 algorithms detector, SIFT is applied to describing keypoint features.
- Defines `SIFT_descriptor` function computes SIFT (from OpenCV's library) keypoints and descriptors for two input images.
- Then, identifying and drawing `good matches` (implement in **6**) between these images based on their descriptors.
- The resulting visual representation showcases matched keypoints using OpenCV's `drawMatches` function.

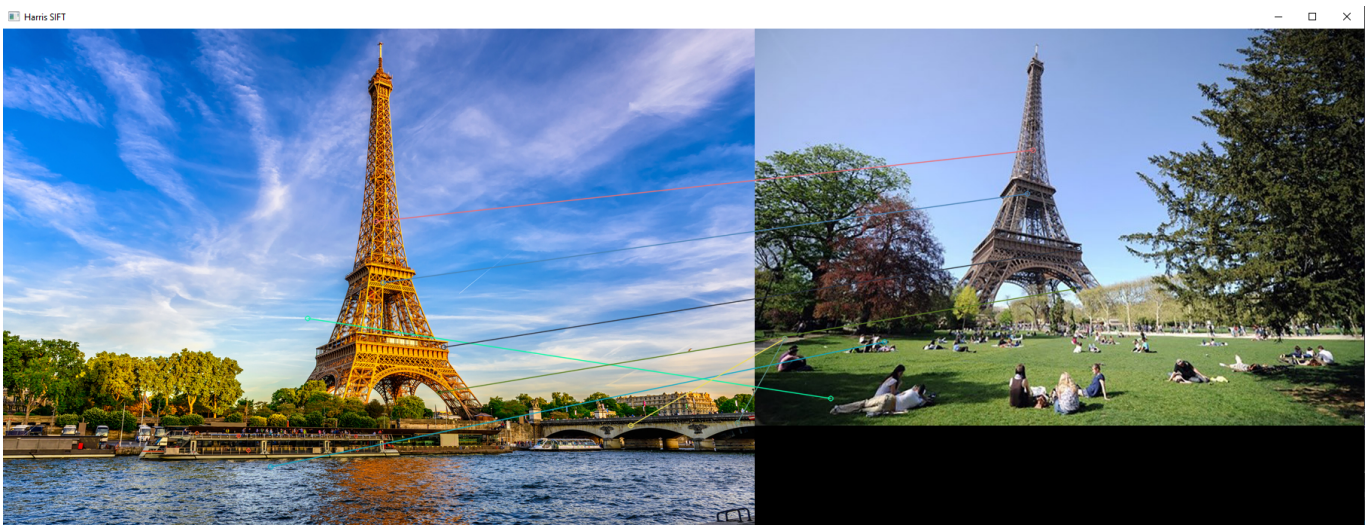


Figure 4: Harris SIFT

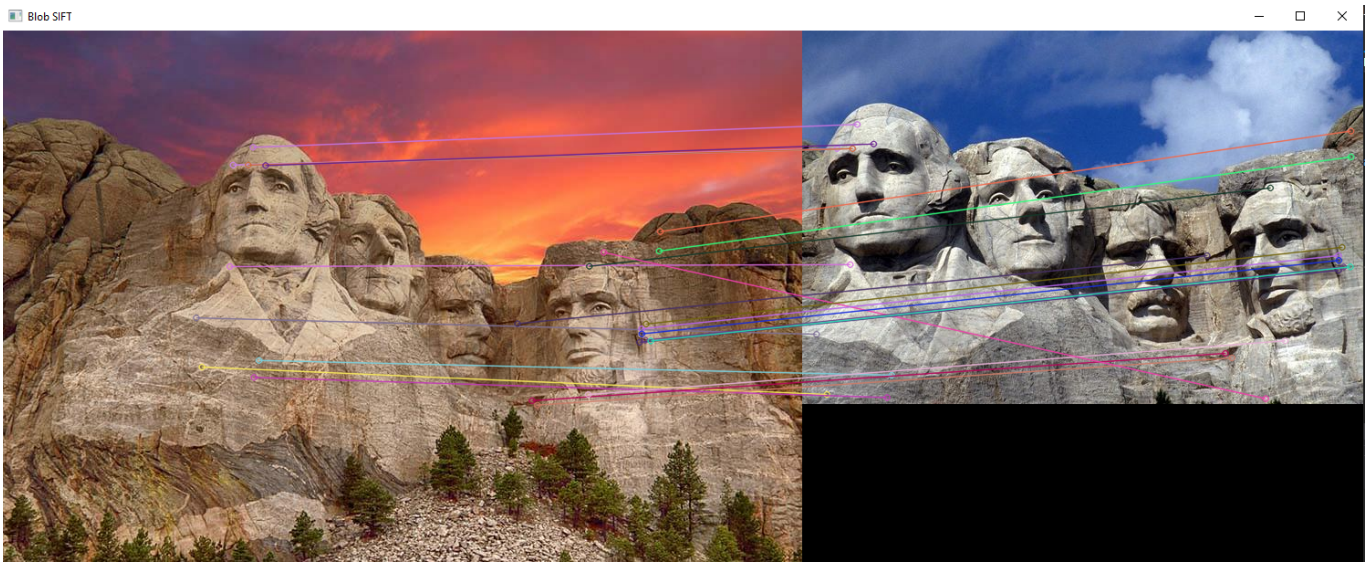


Figure 5: Blob SIFT

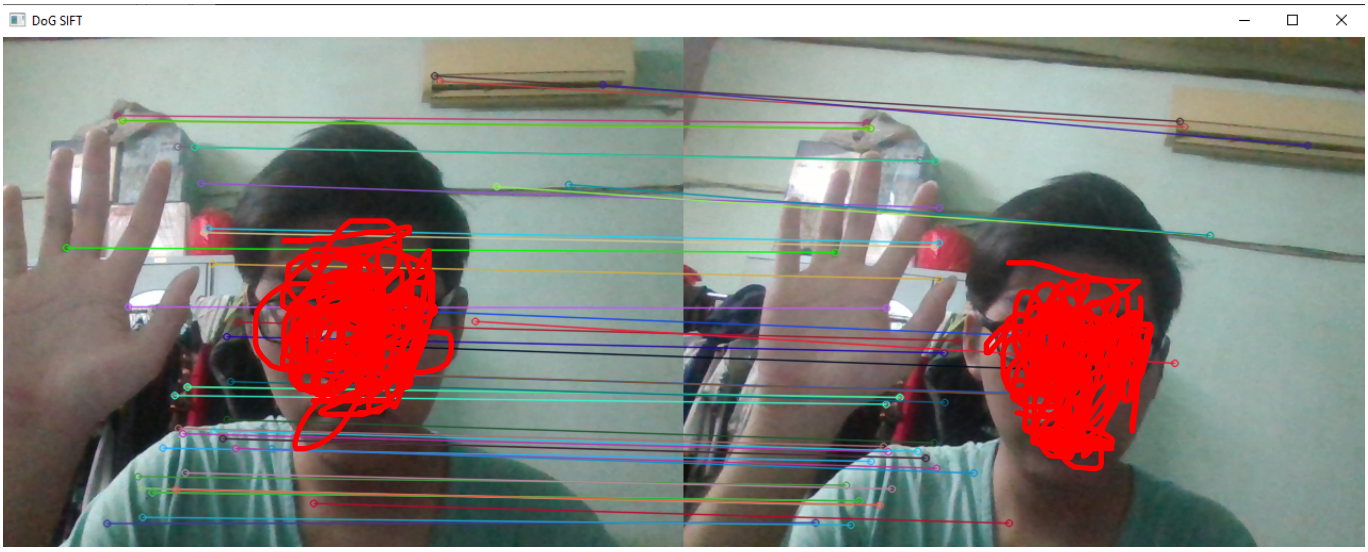


Figure 6: DoG SIFT

5. LBP Descriptor

- Source: LBP.cpp
- The `calculateLBP` function computes LBP value for a given pixel location (x, y):
  - In my implementation, the binary string starts from `(-1, 0)` from the center and moves counter-clockwise direction.
  - After that, updates a histogram matrix to count the occurrences of different LBP. The LBP `[0, 255]` is used as an index to increment the count in the histogram.
- The `computeLBP` function computes LBP descriptors for `keypoints` detected in grayscale images, accumulating these descriptors for subsequent comparison:
  - For valid keypoints (coordinates are within the valid range), it calls the `calculateLBP` function to obtain the LBP descriptor for that keypoint's neighborhood.
  - All individual LBP descriptors in the descriptors vector are concatenated into a single matrix using `vconcat`.
- Applying `find_good_matches` function to draw keypoints similar to SIFT.

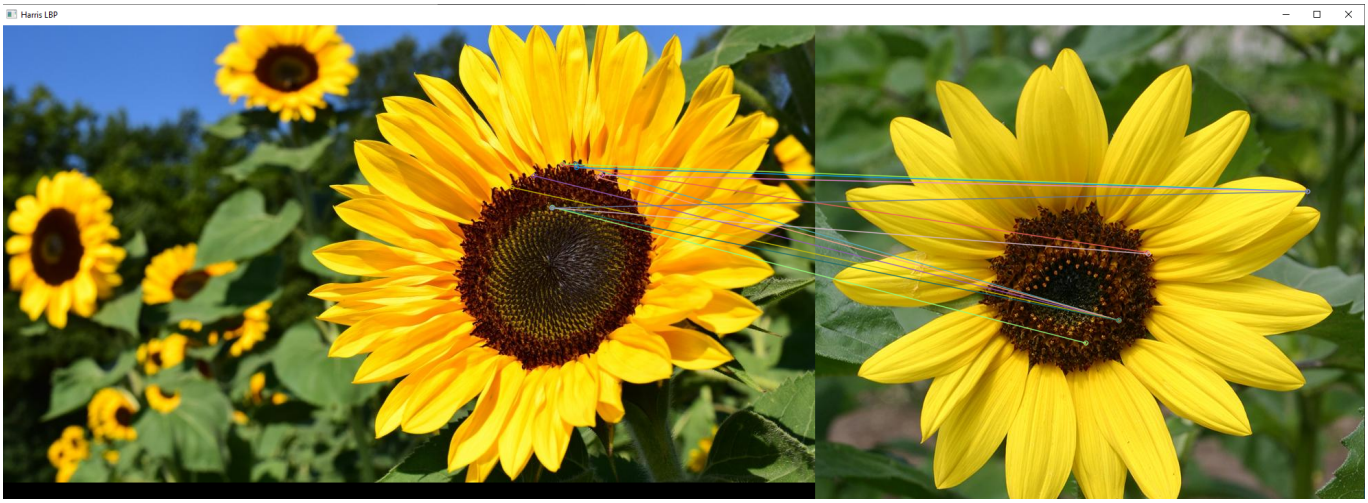


Figure 7: Harris LBP

Figure 8: Blob LBP

Figure 9: DoG LBP

6. Finding Good-Match Keypoints

- Source: Good_Matches.cpp
- Define `find_good_matches` function that uses a descriptor matcher (KNN match) where `FLANNBASED` is used for based matcher to find potential matches between descriptors and keypoints extracted from two input images.
- Applying a ratio `threshold = 0.65` to filter matches, considering only those where the distance to the closest neighbor is significantly lower than the distance to the second-closest neighbor.

III. Evaluation

I selected 5 objects including `butterfly`, `Ben Thanh market`, `Eiffel tower`, `Rushmore mountain`, and `sunflower` to evaluate the performance of detectors and descriptors.

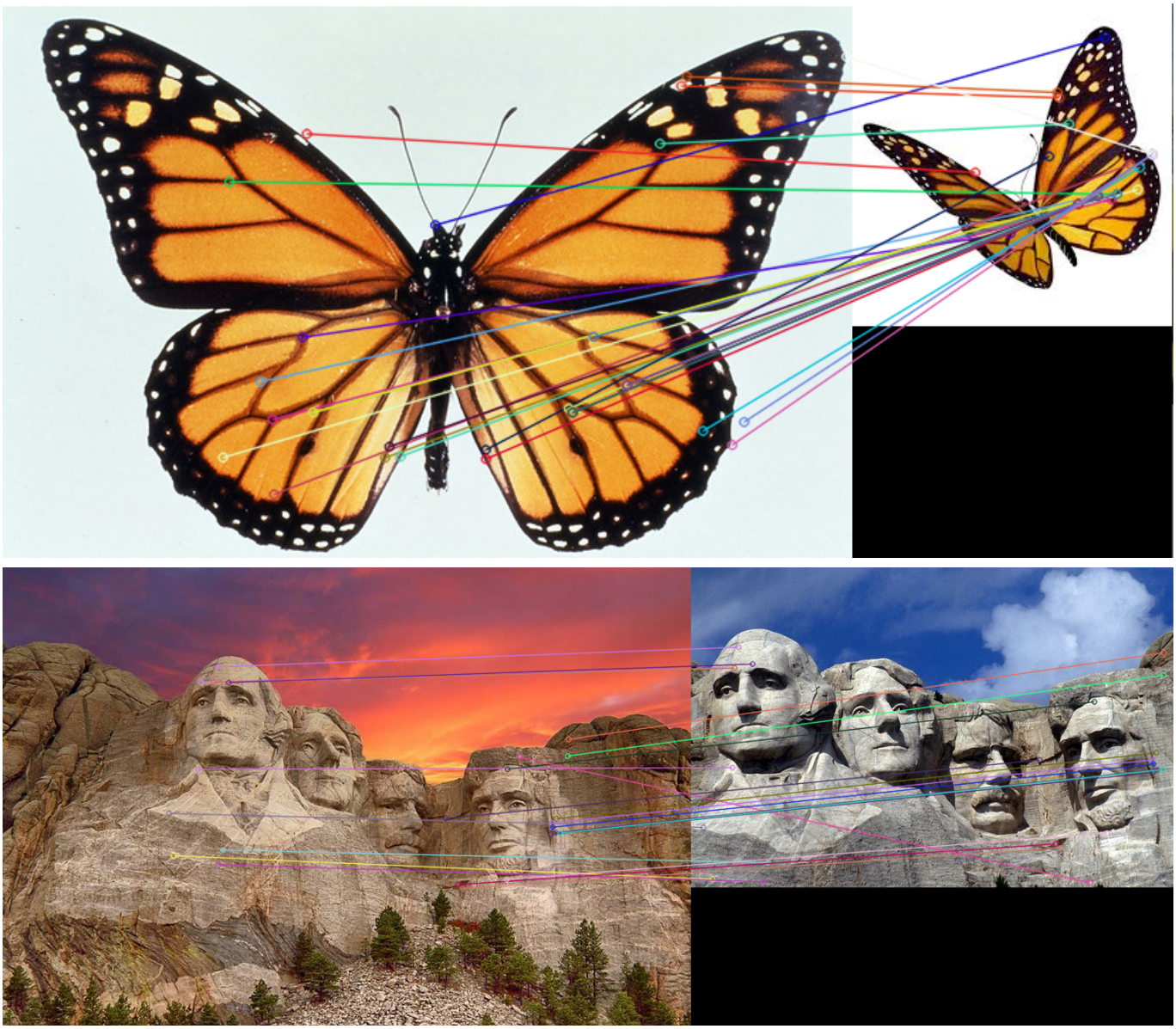


Figure 10: SIFT's performance with 3 different detection algorithms

Figure 11: LBP's performance with 3 different detection algorithms

	SIFT	LBP
Strength	Non-dependance on keypoints. Robustness to scaling and rotation.	Simple and fast. Invariant to bright changes.
Weakness	Memory consumption. May cause noise matching if the number of keypoints are large.	Need to optimize parameters. Dependance on keypoints. Limited discriminative.