**CS 763/CS 764: Lab 04: What's interesting?**

**Features**

- Announced 05/02. Due 13/02 22:00
- Please write (only if true) the honor code. You can find the honor code on the web page. If you used any source (person or thing) explicitly state it.
- **This is NOT an individual assignment**

# 1 Overview

Most 'upstream' vision applications (such as a robot navigating in a warehouse) depend on obtaining interesting points, or keypoints as discussed in the lectures. This is because we don't want to carry several megapixels of information with us that a camera provides (this is also true for human visual system; when you look at a picture, you may automatically pivot to the face to check if the person is smiling).

Once interest points (or keypoints) are found, they need to be compared to the photo you have stored in memory ("is this the same person I saw 2 years ago?"). This is where keypoint descriptors, or simply descriptors come in. It is useful to match these descriptors.

Interest points and descriptors come under the category of "features" in computer vision. The goal of this lab is to explore features.

## 1.1 Part A

This part is a directed assignment involving provided data; specifically pairs of images are considered here.

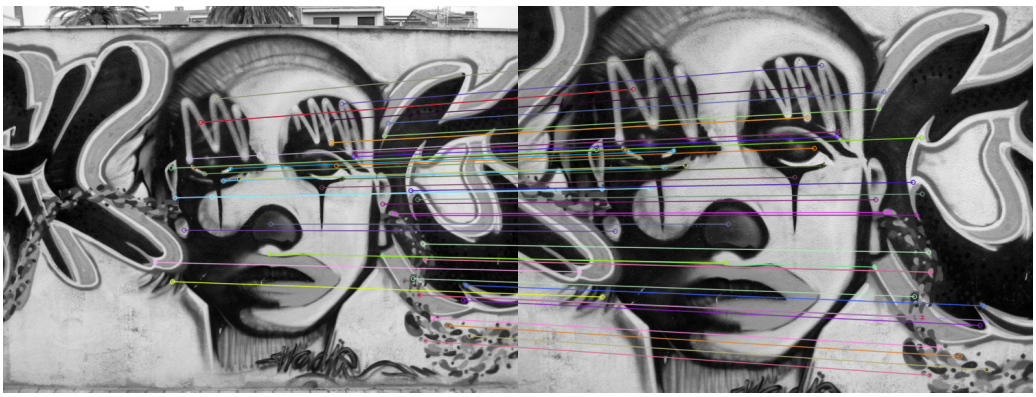

Figure 1: Sample output

Given two photos of the same scene differing in viewpoints or illumination, the goal is to first extract **interest points** from the two photos and, then, to **automatically** establish correspondences between the interest points in the two images.

When you find an interest point in one image corresponding to an interest point in the other image, you show that result by drawing a line between those points in an image in which you show the two photos juxtaposed, i.e., next to each other.

In this part we use two popular detectors viz., (i) **FAST** detector, and (ii) **Difference of Gaussian (DoG)** detector. For FAST, we will use the implementation that comes as the detection part of ORB. For DoG, we will use the implementation that comes as the detection part of SIFT. Once the interest

points in two images are detected, feature descriptors corresponding to each point is computed to encode the information around the interest points into vectors. There exists several methods to compute a descriptor, in this part we use (i) **SIFT**, and (ii) **BRIEF**.

Note that we use use SIFT twice, once for detection of interest points and once for encoding descriptors.

Finally, corresponding pairs of interest points are matched using a notion of proximity in a suitable metric space.

### 1.2 Tasks On Synthetic Images

You are provided with multiple pairs of images. For example, a pair might contain images of the same scene taken from a slightly different viewpoint. The pairs of images will be provided in the `data` folder. The images have the prefix 'view', 'scale', 'rot', and 'light' indicating (typically) synthetic transformations. In other words, they are not "in-the-wild" images, but synthetically created photometric and geometric transformations. You should not be misled by extrapolating these results to real world images except to know the indicative nature of these experiments. 'S', 'T' in the data folder indicate source and target images, i.e., the goal is to match source features to target features.

We use OpenCV's implementation of detector and descriptor algorithms in our task. Unfortunately SIFT is not bundled in the latest OpenCV. Therefore you may need to backdate yourself to an earlier OpenCV version viz. version 3.4.5.20 and 'opencv-contrib-python' with version 3.4.2.17. Other combinations may also work and it is fine to go ahead with those. However, make sure you use a separate virtual environment so as to not clutter your system with incompatible versions.

Run your program as so where arguments take the values shown (without quotes):

```
python features.py -kp [fast|dog] \
-des [sift|brief] -trans [view|scale|rot|light] -nm matches
```

All relevant combinations apply for full credit. `matches` will default to 50 if not specified.

Inside the script the following operations are to be done:

1. Read the two images (with suffix 'S' and 'T') based on the argument '-trans'.
2. Repeat the following steps twice such that the number of interest points (denoted by `nPoints`) in each image is approximately 300 and 1000 in the two iterations respectively.
   (a) Extract interest points from both images based on the argument '-kp'
   (b) Compute descriptors given the interest points. You must use the default options while creating the descriptors.
   (c) Use OpenCV's brute-force matcher to match.
   (d) Draw (using `drawMatches()`) top `matches` matches between the two images based on the relevant metric with lines joining the matched points as shown Fig 1.
   (e) Save the resultant image in the `results` folder with the name `kp_des_trans_nPoints.png`

#### 1.2.1 Questions on Synthetic Images

Whenever you encounter a choice, use lexicographic order (keypoint, descriptor, transformation) to write your answer to avoid penalties. Explain where necessary briefly.

1. For fixed `nPoints` list the **valid** combinations possible. In each case,
   (a) Mention the parameter values used in the detectors to achieve the desired number of detected points.
   (b) Mention average computation time in seconds (`*.xx`)

2. Mention the specs and OpenCv version of your machine used in the previous step to report performance.

3. For (top) `matches=5` will 5 lines be visually seen? Explain.

4. Provide an image (dog, sift, ?) named `incorrect.png` which shows 3 incorrect matches. Number and highlight these, and explain your answer. (You can use any image annotation tool of your choice but each group member will do one each, and name the tool used).

5. When using FAST interest points, which descriptor shows better performance in matching interest points on an average (across all the pairs)?

   (a) What can be an advantage of the inferior descriptor method over the superior?

   (b) In which of the 4 transformation the difference in performance between the superior and inferior descriptors is the maximum and why?

### 1.3 Tasks On Real Images

Instead of working with the provided synthetic images, create pairs of images with photos taken by your own camera. Your pairs should mimic the synthetic examples provided. Next, run the same set of experiments but with a single (detector, descriptor) of your choice on all the images. The (detector, descriptor) combination should be one in this list and other than the ones we have covered.

#### 1.3.1 Questions

1. Explain your choice of (detector, descriptor).

2. Write a short note (two paragraphs at least) on the observations and inferences.

3. Let's assume we are stuck with a detector and a descriptor given to us. What is under our control is the matcher, and assume we are concerned with accuracy. Suggest heuristics that can be used to prune out 'false' matches.

## 2 Part B: Who's the intruder?

Can your surveillance system identify an intruder if he comes to your room masked? Let's experiment with your masks!

Here is the format of the experiment. We are going to pretend that one of your group members in the list Low, Middle, High is the intruder but (s)he is going to come masked. You are given a database, `./replicate/database_image` of faces. Important Note: Please do not redistribute or otherwise tamper with this dataset.

1. Take new snapshots of each of your faces, and add these to the database.

2. It may be difficult to spot a masked intruder. To make this task easier, we will allow you to calibrate. Take a picture `maskLow` of `Low` with mask on. Using this image and the database of images, obtain a reference retrieval score with which you can be confident that you are able to report a true positive, and minimize false positives. Here's an idea for calibration. Suppose we normalize the match score from 0 to 1, with 1 being a certainty. If we set the reference score as 0.01, given `maskLow`, we will say "yes, the intruder is in the database" because the matcher will almost surely return a score higher than 0.01 with some random person in the database. False positives value is very large. If we set the reference score as 0.99, then because of the mask, even if the intruder `Low` (one of the group members) is present in the database, there may not be a match. Pick your score but stick to it.

   Store all freshly captured images in `replicate/capturedImages`.

Figure 2: Sample experiment.

3. Now comes the interesting part to answer the question: How much masking is allowed?

   We work with a new second image of `Middle` which is going to be variable. The goal is to optimize on the masking. The second set of images, named `maskMiddle?.png`, should capture image of the same group member with a "mask" defined as follows. (Ideally we want to use bigger and bigger mouth-nose-forehead masks but since we don't want to keep cutting the mask, we will use a paper to simulate this).

   - The mask for this image should be formed by holding a paper in front of the face in the image (see Figure 2 for example).
   - The paper should contain a hole towards the right. If the hole is a pinhole, i.e., the paper completely covers the face except for the pinhole, the retrieval score is expected to be below your reference score. Now, you are allowed to vary the diameter of this hole. Start increasing the size of the hole to allow your face to become increasingly visible so that the retrieval score increases.
   - Report the diameter of the hole made to the paper at which the reference retrieval score is achieved. If there is no paper, then the diameter is $\infty$.

4. Try the experiment with `Low` (i.e., a somewhat contrived experiment, since you have already calibrated with `Low`), if you get unsatifactory results working with `Middle`.

**Note:** You are allowed to manually crop the images

**Steps:**

1. Store all captured images `maskLow.png` and `maskMiddle?.png` (various versions) and add it to the folder `capturedImages`.

2. Change the diameter of the hole in paper forming the mask to achieve retrieval scores that are worse, and as good on the query images `maskMiddle?.png`. Report all scores in `retScores.txt`

3. Save the version of image `maskedMiddleBest.png` for which the above goal is achieved in the folder `output`.

4. Store the matching retrieval scores in a file `retScores.txt` in the folder `output`.

**Run:** `python3 mask-match.py -i maskLow.png -j maskMiddle?.png`

# 3 Submission Guidelines

Submission guidelines generally remain the same. However there are some parts that can change, so be flexible.

Your submission folder should look something like:

```
130010009_140076001_150050001_Task04/
├── ReflectionEssay.pdf
├── brieflySift
│   ├── code
│   │   ├── features.py
│   │   └── real.py
│   ├── data
│   │   ├── light_S.ppm
│   │   └── created
│   │       └── view_S.ppm
│   ├── results
│   │   ├── brief_fast_light_1000.png
│   │   └── created
│   │       └── x_y_view_1000.png
│   └── answersA.pdf
├── intruder
│   ├── answersB.pdf
│   ├── code
│   │   └── mask-match.py
│   └── replicate
│       ├── capturedImages
│       │   ├── mask_Low.png
│       │   └── mask_Middle1.png
│       └── output
│           ├── mask_MiddleBest.png
│           └── retScores.txt
└── readme.txt
```