Week 2: SIFT and its applications

Efstratios Gavves [*]

November 8, 2013

1 Summary

1.1 Provided:

- Handout
- Template report
- Basic code and functionality
- Dataset

1.2 Requested:

- A report. The report should be written according to the template that is provided in the package you downloaded. **The questions which you need to answer in the report will be marked with red font in this handout.** The report should be delivered after week 2, including the assignments of week 1 and week 2, at `efstratios.gavves@gmail.com`. The **deadline** for delivering report 1 (for the assignments of week 1 and 2) is Wednesday 13th of November, 23:59:59. Delay in delivering the report will be penalized with **minus 1 point per day**.

- A zip file with the implemented code. The places where you are suggested to put your implemented code (this time in *week2.py*) will be marked with comments

  `[# WRITE YOUR CODE HERE], or something similar.`

  You **should not** write code where there is a comment

  `[# DO NOT TOUCH]`

  We do so to ensure that the final code will look as clean as possible. That way we will use our time in evaluating your code and giving you feedback, and **not debugging it** or trying to understand it. We are not responsible for debugging, please make sure the code you send us works. If there are bugs you will be asked to re-submit once **at most**.

[*]Email: `efstratios.gavves@gmail.com`, Room: C. 3.244, Office hours: Mon 17.00-18.00

1.3 Useful links:

- `https://www.enthought.com/` Canopy Enthough environment
- `http://effbot.org/imagingbook/pil-index.htm` Python Imaging Library Handbook
- `http://docs.scipy.org/doc/` Python Numpy and Scipy Library documentations
- `https://piazza.com/uva.nl/fall2013/uva/home` Piazza.com, the electronic embodiment of our class, where you can ask questions and we will answer. Asking questions here is good, as everybody can profit from the answer.

2 Feature detection and description

2.1 Summary

In the second week's assignment we will get more familiar with the SIFT features, one of the state-of-the-art features in computer vision. Due to their stability, descriptiveness and robustness to noise, SIFT features are widely used in a number of visual applications, such as object recognition, image stitching, image reconstruction etc. This week we will focus on

- Understand the SIFT feature, its strengths and weaknesses
- Use SIFT features to compare different images, similar to the first week's assignment
- Use SIFT features to compute the geometric relation between images of the same scenery
- Use these geometric relations to create a panoramic image, which will a collage of the related images.

2.2 Part 1. Download the provided package for Week 2

First download the assignment package from

`http://staff.science.uva.nl/~gavves/files/multi_week2.zip`.

Extract the package in your working directory $main_dir, like in the previous week e.g.,

```
C:\Users\username\multimedia_course
```

Before any kind extraction, please **make a backup** of your previous code. Taking a backup is essential, better be safe than sorry.

2.3 Part 2. SIFT features

2.3.1 Step A. Compute SIFT features

Read the image `../../data/oxford_scaled/all_souls_000026.jpg`. Then run the already provided function

```
impath1 = '../../data/oxford_scaled//all_souls_000026.jpg'
frames1, sift1 = week2.compute_sift(impath1)
```

This function uses an external binary from the VLFEAT library [1] to compute the original SIFT features.

Firstly, it outputs the frames, that represent the locations of the SIFT features together with their scale and orientation. The first two rows are the x, y coordinates, the third one is the scale of the features (how big the circle around (x, y) is) and the forth one is the orientation of the SIFT feature.

Secondly, it also outputs the respective SIFT feature. Each SIFT feature has 128 dimensions. SIFT features are stored row-wise.

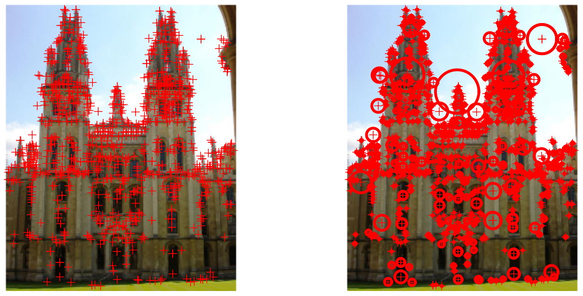

Figure 1: SIFT features.

2.3.2 Step B. Plot SIFT features

Next, plot the detected features from SIFT on the image. To do so, you need run the function `plot_features`,

```
week2.plot_features(im1, frames1, False, 'r')
```

The input arguments for the function should be:

- The *first* input for the function is the image.

- The *second* argument of the function is the frames, as returned from the `compute_sift` function.
- The third argument specifies whether the SIFT feature will be plotted as a circle or just as a point. If it is printed as a circle, the radius of the circle will be equal to the scale of the respective SIFT feature, thus giving an impression of the patch size, over which SIFT was computed.
- The last arguments just specifies the color of the visualization.

In the end, if you have implemented correctly the function, you should get pictures like Fig. 1

2.3.3 Step C. Match SIFT features

Next, we will try to match SIFT features between different images. Matching means that we will define correspondences between similar SIFT features in two images. Open two images from oxford and computer their SIFT features, that is

```
impath1 = '../../data/oxford_scaled/all_souls_000026.jpg'
frames1, sift1 = week2.compute_sift(impath1)

impath2 = '../../data/oxford_scaled/all_souls_000068.jpg'
frames2, sift2 = week2.compute_sift(impath2)
```

Step C.1 Normalization In the assignment of week 1 you had to rank images according to their color histograms. As you may have noticed, the top ranked image was not always the image itself. This comes as a surprise, as one cannot expect to find something more similar to an object, than the object itself.

The reason for this phenomenon was that the features were not normalized. The normalization re-calibrates the values in the histogram (of the feature). That way all histograms are weighted the same, hence we avoid that some histograms have a "stronger" impact for the wrong reasons, for example because we have a bigger image or sharper contrast. Normalization is often crucial for many computer vision algorithms.

Although there are several normalization procedures, we are going to present here two basic ones that couple well with the distance measures you learned about in week 1, see Table 1.

The ℓ_1-normalization takes as input a D-dimensional histogram h and normalizes it like

$$\hat{h} = \frac{h}{\sum_{i=1}^{D} h_i}, \tag{1}$$

such that in the end if you sum up all the elements in h, one should get 1.

The ℓ_2-normalization takes as input a D-dimensional histogram h and normalizes it like

$$\hat{h} = \frac{h}{\sqrt{\sum_{i=1}^{D} h_i^2}}, \tag{2}$$

such that in the end if you compute how long is the vector $\hat{h}$ (`numpy.linalg.norm` *in Python*), one should get 1.

Normaliz./Distance	Euclidean	ℓ_2	Hist. intersection	χ^2-distance	Hellinger
ℓ_2 normaliz.	✔	✔	-	-	-
ℓ_1 normaliz.	-	-	✔	✔	✔

Table 1: Normalization procedures.

Now, before comparing the SIFT vectors you computed above, you need to normalize them using the provided functions `tools.normalizeL1` and `tools.normalizeL2`

```
sift1 = tools.normalizeL1(sift1, DEFINE_AXIS)
sift2 = tools.normalizeL1(sift2, DEFINE_AXIS)
```

or

```
sift1 = tools.normalizeL2(sift1, DEFINE_AXIS)
sift2 = tools.normalizeL2(sift2, DEFINE_AXIS)
```

depending on which distance measure you have.

The second argument `DEFINE_AXIS` of these functions defines the axis of the normalization. If you set `DEFINE_AXIS=0` we normalize per column, whereas if you set `DEFINE_AXIS=1` we normalize per row. **Make sure that you normalize per histogram, and not per dimension of the histogram.** Please, report the difference in your report.

Step C.2 Compute distance between normalized SIFT features and match. In week's 1 assignment we saw various distance measures. We normally use the ℓ_2 distance function to measure how similar to SIFT features are. However, you are advised to experiment with the various distance measures and keep the one that you believe works best. **Make sure** that you first normalize the SIFT features according to Table 1.

Measure the distance between the normalized `sift1` features and `sift2`. Then, for each vector $sift1_i$, find the two closest vectors $sift2_j$ and $sift2_k$, the same way you did it for *Week 1.* If $sift1_i$ is much closer to the closest other, $sift2_j$, than to the second closest, $sift2_k$, it means we have a reliable match. This is the ***Lowe's match criterion*** proposed by the creator of SIFT, David Lowe. You can program this criterion it according to

$$\frac{distance(sift1_i, sift2_j)}{distance(sift1_i, sift2_k)} < thres \tag{3}$$

where `thres` is a threshold number that you can tune. Note, if the distance says that the larger value the closer the histograms, you need to reverse the direction of the inequality to $>$.

Now, you only need to loop for all i, that is for all the SIFT vectors in $sift1$, and find all possible matches that satisfy eq. (3). You can implement this first in the sandbox, then inside the function `week2.match_sift`. *The function is already partly implemented, you just need to finish it.* When you call the function, it should look like

```
matches = week2.match_sift(sift1, sift2, 1.1)
```

This function takes three inputs.

- The first two input arguments are the SIFT features from image 1 and 2.
- The third input argument is a threshold, that defines how strict our matching should be.

The output of the function should be a list of indices that declares which feature from `sift1` matches with which feature from `sift2`. It should contain as many elements as `sift1`. For those features from *sift1* that no matches were found, the respective entry in `matches` should be -1. For example, if the `sift1` contains 8 sift vectors, then if we get that

```
matches = [-1, 4, -1, -1, 2, -1, -1, -1],
```

it means that the 2nd feature from `sift1` is matched to the 5th feature from `sift2` *(remember, in Python we start counting arrays from 0, so 4+1=5)* and the 5th feature from `sift1` with the 3nd feature from `sift2`, etc. The rest features from `sift1` were not matched, hence for these we have -1 entries.

After having implemented the `match_sift` function, visualize the matches using the provided function `plot_matches`. Pick two similar pictures of your choice and match their SIFT features for different values of thresholds. Save the visualizations and paste them into the report. The stricter the threshold, the fewer matches there should be. Finally, report the threshold you are going to use for the rest of the assignment.

If your implementation is correct, you should get a result like in Fig. 2. The figure was generated using the ℓ_2 distance (so the direction of the inequality in eq. (3) changes) and a threshold 1.1.

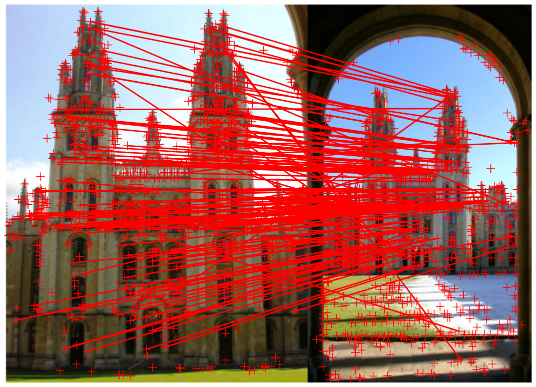

Figure 2: SIFT matches.

2.4 Part 3. SIFT features are invariant to ...

An important concept in computer vision and object recognition is invariance. Invariance to a certain phenomenon, is when our algorithm is not affected by that phenomenon. In general, we want our algorithms to be invariant to various sources of noise that occur in real images. For example, when we compute some representation for one image, then zoom-in in the image, we want the new representation after the zooming-in to be as similar as possible to the original one. This is called **scale invariance**, because no matter what is the scale of the details in the image, we are able to detect them, within some practical limits of course. SIFT features are particularly popular because of their invariances.

If a feature is invariant to a *variation X*, then we expect that we will discover **the same** features, even when the image is affected by the *variation X*. Therefore, it is reasonable to expect that the matching of features will not be affected by the presence of the variation X. Hence, in the end we will approximately retain the same number of matches.

We will elaborate more on 4 cases and in the end you should answer for which of these cases SIFT is invariant. In all these four cases we will visualize the matches and count them. **You don't need to implement any function here, only use the provided code and the `match_sift` function from above**. When SIFT is invariant to a certain variation, the number of matched features should not change much.

2.4.1 Step D. Rotation

Open an image and calculate its SIFT features.

```
impath1 = '../../data/oxford_scaled/all_souls_000026.jpg'
im1 = Image.open(impath1)
frames1, sift1 = week2.compute_sift(impath1)
```

Then, rotate the same image about $15°, 30°, 45°, 60°, 75°, 90°$, using the function `im1.rotate(deg)`, where *deg* stands for the angle of rotation.

Count and plot the matches between the original image and each of the rotated ones (that is, in the end we will have 6 visualizations). Make a plot, where in the x-axis you have the degrees of rotation and in the y-axis you have the respective number of matches. Based on this plot answer and explain in your report whether **SIFT is rotation invariant**, that is whether the number of matches fluctuates significantly for different rotations?

2.4.2 Step E. Scale

Open an image and calculate its SIFT features.

```
impath1 = '../../data/oxford_scaled/all_souls_000026.jpg'
im1 = Image.open(impath1)
frames1, sift1 = week2.compute_sift(impath1)
```

Then, scale the same image using the function `im1.resize(width, height, Image.BICUBIC)`, where *width*, *height* are the new width and height for the image. Retain the same width to height aspect ratio, that is $r = (\alpha width)/(\alpha height) = constant$ and vary the value of $\alpha = 0.2, 0.5, 0.8, 1.2, 1.5, 1.8$.

Count and plot the matches between the original image and each of the scaled ones. Make a plot, where in the x-axis you have the different scales and in the y-axis you have the respective number of matches. Based on this plot answer and explain in your report to what degree **SIFT is scale invariant**, by checking how does the number of matches fluctuate for different scales?

2.4.3 Step F. Perspective

Open an image and calculate its SIFT features.

```
impath1 = '../../data/oxford_scaled/all_souls_000026.jpg'
im1 = Image.open(impath1)
frames1, sift1 = week2.compute_sift(impath1)
```

Then, compare this image with the same image after it underwent a perspective transformation. You can find these images in the `data/other` directory, as you will see also in the provided code.

Count and plot the matches between the original image and each of the ones with change in perspective. Make a plot, where in the x-axis you have the different perspectives and in the y-axis you have the respective number of matches. Based on this plot answer and explain in your report whether **SIFT is perspective invariant**, that is whether the number of matches fluctuates significantly for different perspectives?

2.4.4 Step G. Brightness

Open an image and calculate its SIFT features.

```
impath1 = '../../data/oxford_scaled/all_souls_000026.jpg'
im1 = Image.open(impath1)
frames1, sift1 = week2.compute_sift(impath1)
```

Then, change the image brightness using the `enhance` function from the `ImageEnhance` library. Use the following values $\beta = 0.5, 0.8, 1.2, 1.5, 1.8$ to change the brightness.

Count and plot the matches between the original image and each of the ones with different brightness. Make a plot, where in the x-axis you have the different levels of brightness and in the y-axis you have the respective number of matches. Based on this plot answer and explain in your report whether **SIFT is brightness invariant**, that is whether the number of matches fluctuates significantly for different brightness levels?

2.5 Part 4. Using SIFT for geometry

We live in a 3D world. Hence, in principle the points that we see in two different pictures of the same scenery should be somehow connected in the actual 3D space. It turns out that they are connected indeed when the scenery is planar, and that we can discover this connection in the form of a 3x3 matrix, which we refer to as the homography matrix, that is

$$H = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & 1 \end{bmatrix} \tag{4}$$

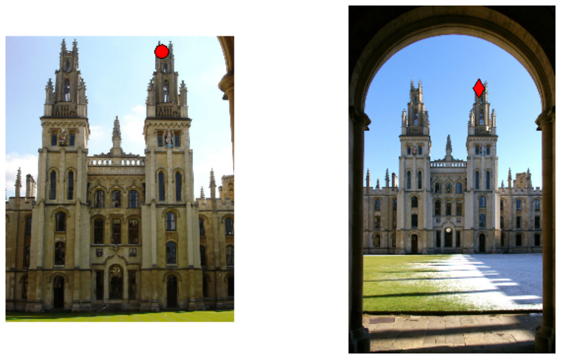

Figure 3: Finding and projecting points in the new image. We set the point in the left image(red circle) and we automatically compute its location on the right image (red diamond).

As we see the homography matrix has 8 degrees of freedom, as we need to define 8 variables to have a valid homography. The homography matrix can be furthermore simplified, *when we don't expect very strong perspective transformations between images*. In that case $h_7 = h_8 = 0$, in which case we have a simpler problem as we need to only define 6 degrees of freedom (also known as *affine transformation*). When we do have strong projective transformations, this simplification will be quite inaccurate. For now we will stick with the full homography matrix.

The homography matrix is computed **between 2 images** of the same scenery. Since the homography matrix has 8 variables we need to estimate. Since each point has 2 values, row and column, we only need 4 pairs of points between the two images to end up with 8 point-to-point relations, that would allow us to solve for the 8 variables of the homography matrix.

2.5.1 Step H. Find the points

After having computed the homography matrix, if we have a point from the image and want to find where it lies in the second image, we just need to perform the following matrix-vector multiplication

$$x' = Hx \tag{5}$$

The better the homography matrix is, the more accurately the new point will be located in the second image.

Open the two images

```
impath1 = '../../data/other/all_souls_000026.jpg'
frames1, sift1 = week2.compute_sift(impath1)

impath2 = '../../data/other/all_souls_000068.jpg'
frames2, sift2 = week2.compute_sift(impath2)
```

We provide the homography matrix H for this pair of images. Then, you need to implement the remaining of function `week2.project_point_via_homography` by just including the matrix to vector multiplication from eq. (5). If you have a correct implementation, you should get something like Fig. 3.

Please, define yourselves 5 points and project them on the second image with different colors. Plot all the projected points to the same image. Make sure that you implemented the function correctly, so that to verify that the points were automatically, and not manually, plotted. Save the two images with the original points and with the projected points. Paste them to the report.

2.6 Part 5. Computing homography using RANSAC

To compute the homography matrix, we can consider two ways for the time being.

The first method is the manual one, where the user needs to provide with 4 pairs of corresponding points between two images, telling that these 4 points in one image are the same as some other 4 points in the second image. Apparently, this method is not very practical when we want to have **automatic** object recognition, where the computer should recognize an image without any kind of human intervention.

The second method is RANSAC. RANSAC is a statistical algorithm, that simply put works as follows:

1. First, sample 4 point-to-point correspondences from the available matches we have computed
2. Calculate a homography matrix using these correspondences
3. Evaluate how good your homography matrix is. Evaluation is done by measuring the number of initial matches that agree (we shall call them **inliers**) with the computer homography.

Obviously, by sampling random correspondences, most of the computer homographies will be bad, meaning that their will have very few inliers. If we repeat the above procedure enough times, then it is quite likely to get a good homography matrix, *if there is one*. If the two images are from different sceneries, or if they are two different due to the perspective transformations or other phenomena, RANSAC will likely not return any good homographies.

2.6.1 Step I. Compute homography between 2 images

The code for computing a homography between two images is provided, that is

```
H, inliers = homography.estimate_homography_with_ransac(frames1,
frames2, matches),
```

where `frames1` are the SIFT locations for image 1 and `frames2` are the SIFT locations for image 2. The third argument is `matches` and you can calculate them with the function you have implemented above.

The output of the above image is the homography matrix H defined above, that connects the two images geometrically. The second output of the image are the inliers after the geometic comparison. Inliers contain the subset of the matches which agree with the discovered homography H. The inliers are stored as the indices of the matches (3rd input argument).

Read two images and compute their SIFT features.

```
impath1 = '../../data/oxford_scaled/all_souls_000026.jpg'
frames1, sift1 = week2.compute_sift(impath1)

impath2 = '../../data/oxford_scaled/all_souls_000068.jpg'
frames2, sift2 = week2.compute_sift(impath2)
```

Now, compute the homography matrix between the two and plot the matches which are also inliers. You should get a figure like Fig. 4, where the red lines denote the inlier matches and the blue lines denote the outlier matches.

Repeat for the pairs of images (`all_souls_000026.jpg`, `all_souls_000040.jpg`) and (`all_souls_000015.jpg`, `all_souls_000091_half.jpg`). Paste the result in your report. What do you observe regarding the success of computing the homography?

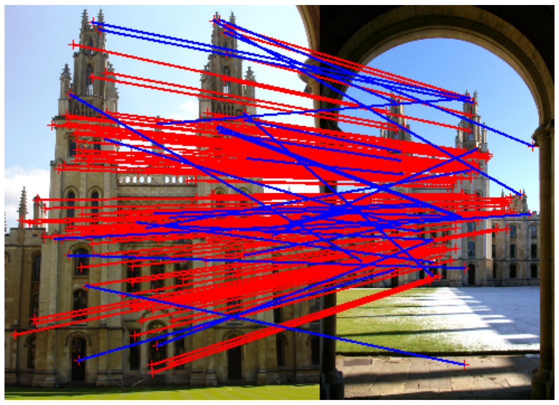

Figure 4: Inlier matches with red, outlier matches with blue.

We will now proceed with application of image recognition using homography-based geometric verification.

2.6.2 Step J. Image recognition with geometric verification

Since we are able to extract the homography, we can now define whether two images depict the same scenery, if they share enough inliers. Read all the files from the oxford directory in the data and test whether two images depict the same scenery or not, by counting the number of inliers. Most of the code is provided, you only need to do the final check whether the inliers are plenty enough or not.

Given the queries `all_souls_000035.jpg` and `all_souls_000040.jpg`, which of the rest images are geometrically recognized. Paste those images, together with the query in your report.

2.7 Part 6. BONUS

1. What is the relation between invariance and normalization? Could we view normalization as bringing some kind of invariance? If yes, what kind of invariance?

2. To project a point from an image I_1 to another image I_2, we need to apply eq. (5) using the homography matrix $H_{1,2}$. Now, if we want to do the reverse, the obvious way would be to re-run RANSAC, matching I_2 against I_1. However, since all points can be expressed in terms of matrix operations, can you find a more elegant, and faster way to compute $H_{2,1}$

$$x = H_{2,1}\, x' \tag{6}$$

without needing to rerun RANSAC?

References

[1] VLFEAT, http://www.vlfeat.org/