**Computer Vision Project**

**Student:** Klim Zaporojets  
**Student Id:** 28057963

**SIFT Descriptor**

**Introduction**

In the present document I will describe my implementation of SIFT description from scratch. The implemented version was made using Matlab. First, I describe the steps I followed in my implementation that are suggested in [1]. Then, I show some of the results achieved by my descriptor. Finally, I make some conclusions and suggestions for further improvements on my work.

**Implementation Description**

**Step I: Detection of Scale-Space Extrema**

The first implemented step consists in detection of scale-space extrema. This is done by calculating the difference of gaussians over different octaves using the following formula:

\[ D(x, y, \sigma) = (G(x, y, k \sigma) - G(x, y, \sigma)) \ast I(x, y) \]

The code was implemented in the class `scaleSpace.m` of my Matlab project. I consider that the following are the main points worth mentioning while detecting scale-space extrema:

1. **Value of K:** this value determines the increment of gaussian blurring with each blurring step. According to [1], the recommended value for k is \( k = 2^{1/s} \) where s is the number of intervals. According to my tests for some images lower or higher values can do best. Usually, when k is higher, the number of keypoints found increases and when it is lower, it decreases. This is because small k values tend to generate very similar scaled images. Therefore, their DOG are also very similar, which generates less keypoints. With big values of k, the DOG images are very different and, in consequence, the high number of keypoints generated in this way are less meaningful and repeatable of a particular position in the image. In this work I opted to use the recommended key value of \( k = 2^{1/s} \).

2. **Kernel Size:** in David Lowe papers ([1] and [2]) it doesn't make any specific reference to kernel size that has to be used when blurring the images with gaussian. However, due to the fact that gaussian blur is based on gaussian distribution that accounts for 99.7% of cases in its \( \pm 3\sigma \), it doesn't make any sense in making a kernel bigger than \( 6\sigma \times 6\sigma \). In fact, in [4] the recommended size of kernel is \( 6\sigma - 1 \). However, according to my experimental results, the best SIFT matching happens with a size of kernel of \( (3\sigma - 1) \times (3\sigma - 1) \).

3. **First image resampling and initial pre-smoothing:** the original image has to be pre-smoothed to avoid capturing irrelevant keypoints that are not repeatable, discarding in this way the highest spatial frequencies. In the section 3.3 of [1] it is also recommended to double the size of original image in order to create more relevant keypoints. From my tests, the number of relevant keypoints can increase by a factor of 3 or more by doing it. The value of \( \sigma \) used is for initial pre-smoothing is \( \sqrt{2} \) since it gave me slightly better results than the value of 1.6 recommended in [1].

---

1 After discarding edges and low contrast keypoints.
4. **Number of Octaves**: the number of octaves depends on the initial image resolution, the higher the image resolution, the higher is the number of octaves that can get repeatable keypoints. From my tests on small images (no more than 500 x 500 pixels), more than 4 octaves practically didn't make any difference of the amount of keypoints detected. Therefore, I use 4 octaves throughout this work.

5. **Number of Scales per Octave**: in the section 3.2 of [1] it is recommended to use 3 scales per octave. I also tried using higher values up to 10 without getting any general improvement. Furthermore, higher number of scales tend to decrease the number of relevant keypoints. I consider that this is due to the fact that when k-value is lower, the found keypoints tend to be more "fine-grained" and, therefore, include a lot of points of low contrast that are not repeatable among different image transformations (scaling, rotations, etc.).

6. **Gray color only**: due to the fact that SIFT descriptor works mainly with gradients and brightness intensities, the best and most efficient way to deal with the processed image internally is by converting it to grayscale. This makes the processing and filtering much faster rather than dealing with full RGB color image.

### Step 2: Keypoint Localization

The maxima and minima of the difference-of-Gaussian images are detected by comparing a pixel to its 26 neighbors in 3x3 regions at the current and adjacent scales (DOGs). This is done in the function calculateKeypoints.m. I think that the following are the two points worth mentioning in this section:

1. **Low contrast keypoints detection**: the detection is made by evaluating the values of DOG images and thresholding those under 0.03 (the image pixel values are in range [0,1]). This is something recommended in section 4 of [1].

   The following image gives an idea of the number of keypoints deleted using this method:

   ![Keypoint Detection](image1.png)

   | 3522 keypoints | After processing low contrast: 503 keypoints |

2. **Detection of edges**: the detection of edges is performed by using the Hessian eigenvalues approach proposed in section 4.1 of [2]. In order to calculate derivatives I use the formulation proposed by [5] and the notes from the class. The only point of interest is that instead of using r=10 proposed in [2], I use r=5 due to the fact that in my experiments it allowed to detect more edges without losing the most important keypoints.

---

2 See section "Conclusions and Future Improvements" for further work and improvements of this point.
3. **imfilter with 'replicate':** finally, it is worth mentioning that the gaussian filtering is performed by imfilter operation with 'replicate' option in order not to accumulate "black edges" around the image. This is because, by default, imfilter fills with 0(black) the regions outside the image boundaries. With 'replicate' option on the other hand, input array values outside the bounds of the image are assumed to equal the nearest array border value.

**Step 3: Orientation/Magnitude Assignment**

When implementing the assignment of orientation/magnitude, I mainly followed the section 5 of [1]. The implemented code is located in defineOrientation.m function. Below are the most relevant points of implementation:

1. **Magnitude detection:** the magnitude detection is done for each blurred scale using the following equation described in [1]:
   \[ m(x, y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x, y+1) - L(x, y-1))^2} \]
   Internally, a filtering is performed for horizontal difference with the following filter:
   \[
   \begin{bmatrix}
   0 & 0 & 0 \\
   -1 & 0 & 1 \\
   0 & 0 & 0
   \end{bmatrix}
   \]
And for the vertical difference with the filter:
\[
\begin{bmatrix}
0 & 1 & 0 \\
0 & 0 & 0 \\
0 & -1 & 0
\end{bmatrix}
\]

Finally, the square root of square sum of differences is calculated. In this way, all the operations are executed directly on matrixes without any need of for-loops that can slow down the performance.

Finally, its worth mentioning that this magnitude detection is carried out on the same octave as a particular analyzed keypoint, using the corresponding upper scale of DOG in order to achieve scale invariance.

2. **Orientation Detection**: the following formula from [1] is used for orientation detection:
\[
\theta(x, y) = \tan^{-1} \left( \frac{L(x, y+1) - L(x, y-1)}{L(x+1, y) - L(x-1, y)} \right)
\]

Internally, the results of the filters of matrixes defined in point 1 are used to achieve better performance in terms of processing speed.

3. **Weighting**: a gaussian weighting is applied centered on each keypoint before calculating the magnitudes. The kernel's $\sigma$ is of 1.5 times that of the scale of the keypoint. This is recommended in section 5 of [1] and, basically, it allows emphasize the magnitudes that are located closer\(^3\) to each keypoint.

4. **Orientation Assignment**: after performing the above steps, an orientation histogram of 36 bins is constructed. The highest peaks as well as all those withing 80% of the highest are detected. As a final step a parabola is fit to the neighbors of each peak. For instance if the peak is located in bin 1 with value of 6.5 (magnitude), the value of bin 36 is 6.0 and the one in the bin 2 is 0.5. Then a parabola can be fit to these points\(^4\):
\[
 f(x) = -3.25x^2 + 3.75x + 6
\]

This parabola has its highest peak at $x = 0.57$, which corresponds to $(0.57-0.5)*10 = 0.7$ degrees, which is less than 5 degrees that correspond to the mid value of bin 1; this is because the "left" neighbor of the highest bin has greater impact than the neighbor on the "right". Finally, this value is assigned to the orientation of the keypoint. The same procedure is performed with other bins within 80% of highest orientation bin.

The following image represents the orientations plotted using the plotDescriptor.m function of my code:

---

3 And therefore, allow to characterize better a particular keypoint than the ones located further away.
4 Internally, Matlab polyfit function is used.
Orientations - note that one keypoint can have more than one orientation

**Step 4: Local Image Descriptor**

The local image descriptor is built following the steps in section 6 of [1] with the code located in localDescriptor_v3.m function. I consider that the most important point to mention is how rotational invariance is achieved in my implementation. The main idea is to align the descriptor to the orientation (or orientations) of a particular keypoint. The following illustration taken from [6] depicts this concept:

![Image showing local image descriptor rotation](image)

The local image descriptor is rotated in direction of the keypoint

It's very important to achieve this invariance since it will allow SIFT to locate the same keypoints when the image is rotated. To achieve this, I rotate the orientation and magnitude matrixes corresponding to each scale in the octave where particular keypoint is located. So, if the orientation of a keypoint is 70 degrees right, the orientation and magnitude matrixes are rotated 70 degrees to the left. After that, the new position of the keypoint is calculated using the rotational matrix studied in class:
Finally, the local image descriptor (size of 128 elements) is calculated with previously gaussian-weighted window with $\sigma$ equals to one half the width of the descriptor window. Furthermore:

1. The image descriptor is normalized to unit length in order to be invariant to light change.
2. The values above 0.2 are thresholded in order to reduce the influence of light gradients.

After this procedure, the vector is re-normalized to unit length again, as recommended in [1].

**Step 5: Matching**

The matching is performed using L2 distance technique between the local image descriptors of two images to compare. This is implemented in getMatches.m function. One important point to be mentioned is that when performing matches, the best (the lowest ones) two L2 differences are stored for a particular keypoint. The idea outlined in section 7.1 of [1] is that if these two differences are close to each other, then the chance of being a good match is higher. This makes sense, since if we know that a particular descriptor has more than one similar match on the other image, they can come from the keypoints of the same region with similar characteristics (i.e. L2 distances). The ratio between the two closest neighbors suggested in [1] is 0.8. In my experiments, however, ratios as high as 1.8 have proven also very efficient, increasing the number of correct matches in more than 50% and still maintaining the number of incorrect matches low.

**Some results and examples of how to execute**

In order to execute my SIFT descriptor the function siftDescriptor.m has to be executed. The following two lines have to be changed in siftDescriptor.m function in order to compare different set of images:

```matlab
image1 = imread('quaker_rot1.jpg');
image2 = imread('quaker_rot2.jpg');
```

These two variables take as values the images to compare. During my tests I compared several images I attach with this work. The following are some of the results:

1. Compare scale and perspective with quaker_s1.jpg and quaker_s2.jpg:
2. Compare scale, perspective and light change with quaker_s1_dark.jpg and quaker_s2.jpg:

77 matches detected, some are wrong again

3. Compare rotation with quaker_rot1.jpg and quaker_rot2.jpg:

15 matches detected, only one seems to be wrong. More matches can be detected using higher values of L2 ratio explained in "Step 5" (the used here is 0.8)
Conclusions and Future Improvements

The SIFT descriptor seems to be working very well on fixed images such as the ones showed above. However, when I tried to run it for different facial expressions and environments with multiple similar features repeated, the ratio of error increases dramatically. For instance, consider the following match made by my SIFT implementation:

![Some matches go astray because of similar features](image)

It can be seen how some matches are erroneous because of same background pattern and the hair which is similar along its whole extension. This is something that was discussed on class when analyzing the problem of performing the match using adaptive bin sizes. I consider that other approaches such as those related to distribution field and congealing can be more efficient to solve these problems since they produce a probability distribution on the whole image which makes them less prone to these kind of mistakes.

I also consider that there are some points in my implementation that need of further work and improvement, the following is a brief list summarizing the most important ones:

- **Close keypoints**: so far, the low contrast points have been detected by eliminating the values of DOG images that are lower than 0.03. However, this still doesn't solve the problem of dealing with extrema points that lie close to each other. The approach suggested by [1] to solve this issue consists in using Taylor expansion up to the quadratic terms to find these extremes. The detection and interpolation of neighbor keypoints can help my descriptor to be more invariant to similar features such as described in the example above. This is because a particular keypoint will not only have the information about itself, but also about the neighbor keypoints interpolated into its own structure.

- **Trilinear interpolation when performing rotation of image descriptor**: as described in step 4, in order to get invariance to rotation, the magnitude and orientation matrixes are rotated (using
nearest neighbor interpolation) according to a particular keypoint orientation/s. Going more into the detail, the paper in [1] suggests to perform trilinear interpolation when rotating the image descriptor. This is achieved by not only interpolating on a particular image of magnitudes/orientations, but also taking into account the position of the vertices of the image descriptor itself after the rotation. This interpolation is better since it allows to distribute the values among the bins of the descriptor with less chance to be affected by boundary effects.

- **Speed and performance:** unfortunately, I wasn't able to achieve high speed when calculating SIFT descriptor. All the tested images have less than 500 x 500 pixels, so the process doesn't take too long. This point can be improved by re-writing the slowest functions (such as the one that calculates the keypoints) in languages that are very fast in doing this processing such as C++. Afterwards, these functions can be re-used from Matlab by using tools such as MEX.

**References**


