Assignment #4

Prepared by
Michael Pérez

Computer Vision
CAP 4410

12 November 2019

Florida Polytechnic University

Table of Contents

List of Figures

1. Introduction

1.1 Introduction

My name is Michael Pérez, and I am a senior studying Computer Science at Florida Polytechnic University. This objective of this project was to implement the SIFT method of feature extraction and feature description for a sample image, as described by David Lowe [1]. I implemented these methods using OpenCV in Visual Studio, in C++. I tested the method on the test image on Canvas: "blocks_L-150x150.png".

1.2 Running the Project

There are two windows that appear immediately when the program is run:

1. Program Console
2. Image (image file name specified in main.cpp)

When the SIFT method is applied to an image, the outputs of step 1 and step 2 of SIFT are displayed in two new windows, sequentially:

3. Scale-space Extrema Detection
4. Key Point Localization

While the program is running, pressing the following button corresponds to this command:

'1': Apply the SIFT method and display the output at each stage.

1.3 Product Scope

This product has a wide range of applications. SIFT can be applied to face recognition, iris recognition, ear recognition, fingerprint recognition, and real-time hand gesture recognition as well. [2] Once edited to have this additional functionality, this program can have many potential benefits. However, limitations in time prevented me from finishing the project.

2. Scale Invariant Feature Transform (SIFT)

2.1 Implementation

I read about the SIFT method in the lecture slides and in the paper by David Lowe, the inventor of the algorithm [1]. I tried over ten different implementations of SIFT that I found on the internet, in both C++ and Python, and ultimately found that source [3] produced the best results. Many implementations used abstract libraries and foreign data structures that I could not figure out how to debug. I edited the code in source [3] substantially, creating the images in steps one and two.

There are four sequential steps in the SIFT algorithm: scale-space extrema detection, key point localization, orientation assignment, and key point descriptor [1].

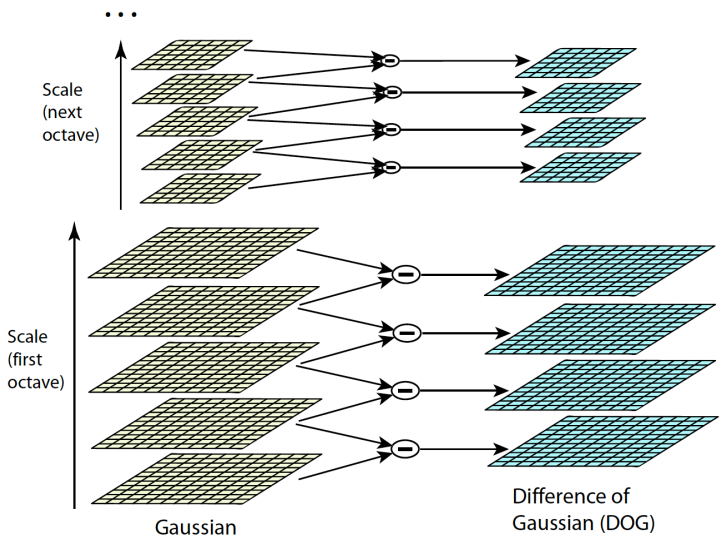

Figure 1: DoG Scale Space Creation

Before the first step, the scale-space of scales and octaves must be created. A two-dimensional array of difference-of-Gaussians (DoG) is created:

```
//Pre-blur Octave 1, Scale 1 image
GaussianBlur(ScaleSpace[0][0], ScaleSpace[0][0], ksize, PREBLUR_SIGMA);

//imshow("after", ScaleSpace[0][0]);
// Populate rest of the Scale Spaces
for (int oct = 0; oct < num_octaves; oct++)
{
    sigma = INIT_SIGMA;  // reset sigma for each octave
    for (int sc = 0; sc < num_scales + 2; sc++)
    {
        sigma = sigma * pow(2.0, sc / 2.0);

        // Apply blur to get next scale in same octave
        GaussianBlur(ScaleSpace[oct][sc], ScaleSpace[oct][sc + 1], ksize, sigma);

        // DoG = Difference of Adjacent scales
        DoG[oct][sc] = ScaleSpace[oct][sc] - ScaleSpace[oct][sc + 1];

        //            cout << "Octave : " << oct << "   Scale : " << sc << "   Sca
    }
```

Figure 2: DoG Scale Space Creation Implementation

Next, in the DoGExtrema() function, for each pixel in each octave and scale of the image, if the pixel is a minimum or maximum (extremum) of the eight pixels around it, the nine pixels in the scale directly above it, and the nine pixels in the scale directly below it, then it is declared a key point:

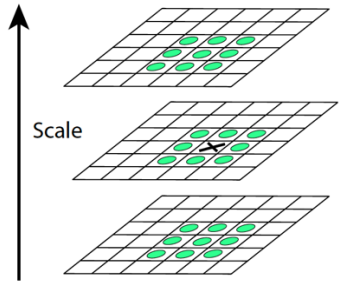

Figure 3: Scale-space Extrema Detection

```
// Look for local maxima
// Check the 8 neighbors around the pixel in the same image
// Range is [lower_bound, upper_bound)
local_maxima = (current(Range(1, sx - 1), Range(1, sy - 1)) > current(Range(0, sx - 2), Range(0, sy -
    (current(Range(1, sx - 1), Range(1, sy - 1)) > current(Range(0, sx - 2), Range(1, sy - 1))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > current(Range(0, sx - 2), Range(2, sy))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > current(Range(1, sx - 1), Range(0, sy - 2))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > current(Range(1, sx - 1), Range(2, sy))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > current(Range(2, sx), Range(0, sy - 2))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > current(Range(2, sx), Range(1, sy - 1))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > current(Range(2, sx), Range(2, sy)));

// Check the 9 neighbors in the image above it
local_maxima = local_maxima & (current(Range(1, sx - 1), Range(1, sy - 1)) > top(Range(0, sx - 2), Ra
    (current(Range(1, sx - 1), Range(1, sy - 1)) > top(Range(0, sx - 2), Range(1, sy - 1))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > top(Range(0, sx - 2), Range(2, sy))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > top(Range(1, sx - 1), Range(0, sy - 2))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > top(Range(1, sx - 1), Range(1, sy - 1))) &  // same
    (current(Range(1, sx - 1), Range(1, sy - 1)) > top(Range(1, sx - 1), Range(2, sy))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > top(Range(2, sx), Range(0, sy - 2))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > top(Range(2, sx), Range(1, sy - 1))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > top(Range(2, sx), Range(2, sy)));

// Check the 9 neighbors in the image below it
local_maxima = local_maxima & (current(Range(1, sx - 1), Range(1, sy - 1)) > down(Range(0, sx - 2), Ra
    (current(Range(1, sx - 1), Range(1, sy - 1)) > down(Range(0, sx - 2), Range(1, sy - 1))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > down(Range(0, sx - 2), Range(2, sy))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > down(Range(1, sx - 1), Range(0, sy - 2))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > down(Range(1, sx - 1), Range(1, sy - 1))) &  // sam
    (current(Range(1, sx - 1), Range(1, sy - 1)) > down(Range(1, sx - 1), Range(2, sy))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > down(Range(2, sx), Range(0, sy - 2))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > down(Range(2, sx), Range(1, sy - 1))) &
    (current(Range(1, sx - 1), Range(1, sy - 1)) > down(Range(2, sx), Range(2, sy)));
```

Figure 4: Scale-space Extrema Detection Implementation

Then, in the FilterDoGExtrema() function, for each scale and octave, each extremum key point is discarded if it is a low contrast point or on an edge. The number of key points decreases:

```
for (int k = 0; k < num_keypts; k++)
{
    x = locs.at<int>(k, 0);
    y = locs.at<int>(k, 1);

    // Discard low contrast points
    if (abs(current(x + 1, y + 1)) < CONTRAST_THRESHOLD)
    {
        DoG_Keypts[oct][sc].at<uchar>(y, x) = 0;
        reject_contrast_count++;
    }
    // Discard extrema on edges
    else
    {
        rx = x + 1;
        ry = y + 1;
        // Get the elements of the 2x2 Hessian Matrix
        fxx = current(rx - 1, ry) + current(rx + 1, ry) - 2 * current(rx, ry);   // 2nd order deriva
        fyy = current(rx, ry - 1) + current(rx, ry + 1) - 2 * current(rx, ry);   // 2nd order deriva
        fxy = current(rx - 1, ry - 1) + current(rx + 1, ry + 1) - current(rx - 1, ry + 1) - current(
        // Find Trace and Determinant of this Hessian
        trace = (float)(fxx + fyy);
        deter = (fxx * fyy) - (fxy * fxy);
        curvature = (float)(trace * trace / deter);
        if (deter < 0 || curvature > curv_threshold)   // Reject edge points if curvature condition
        {
            DoG_Keypts[oct][sc].at<uchar>(y, x) = 0;
            reject_edge_count++;
        }
    }
}
```

Figure 5: Key Point Localization Implementation

At each key point, the magnitude and direction is calculated using local image gradient directions:

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = \tan^{-1}((L(x,y+1) - L(x,y-1))/(L(x+1,y) - L(x-1,y)))$$

Figure 6: Orientation Assignment Formulas

```
Mat_<uchar> current = ScaleSpace[oct][sc + 1];
Magnitude[oct][sc] = Mat::zeros(current.size(), CV_64FC1);
Orientation[oct][sc] = Mat::zeros(current.size(), CV_64FC1);
//cout << current.rows << endl;
//cout << current.cols << endl;
//waitKey(0);

for (int x = 1; x < current.rows - 1; x++)
{
    for (int y = 1; y < current.cols - 1; y++)
    {
        // compute x and y derivatives using pixel differences
        double dx = current(y, x + 1) - current(y, x - 1);
        double dy = current(y + 1, x) - current(y - 1, x);


        // compute the magnitude and orientation of the gradient
        Magnitude[oct][sc].at<double>(x, y) = sqrt(dx * dx + dy * dy);
        Orientation[oct][sc].at<double>(x, y) = (atan2(dy, dx) == PI) ? -PI : atan2(dy, dx);

        cout << "Location[" << x << "][" << y << "]: Magnitude = " << Magnitude[oct][sc].at<uchar>(x, y

    }
}
```

Figure 7: Orientation Assignment Implementation

In the last step, which I did not get to implement due to time constraints, a high-dimensional vector is formed, storing the x and y locations of the key points and the orientations and magnitudes at those locations. This representation of the data would allow for a simple nearest-neighbors algorithm that uses the Euclidean distance between the vectors to check if the difference between two images less than a certain threshold. This would be able to generate a binary output of if two different images have the same subject or not.

2.2 Testing

Figure 8: SIFT Method Test Image

```
Creating Scale Space...


Finding DoG Extrema...
Octave : 0   Scale : 0   DoG_Keypts size : 300x300
Octave : 0   Scale : 1   DoG_Keypts size : 300x300
Octave : 0   Scale : 2   DoG_Keypts size : 300x300
Octave : 0   Scale : 3   DoG_Keypts size : 300x300
Octave : 0   Scale : 4   DoG_Keypts size : 300x300
Octave : 1   Scale : 0   DoG_Keypts size : 150x150
Octave : 1   Scale : 1   DoG_Keypts size : 150x150
Octave : 1   Scale : 2   DoG_Keypts size : 150x150
Octave : 1   Scale : 3   DoG_Keypts size : 150x150
Octave : 1   Scale : 4   DoG_Keypts size : 150x150
Octave : 2   Scale : 0   DoG_Keypts size : 75x75
Octave : 2   Scale : 1   DoG_Keypts size : 75x75
Octave : 2   Scale : 2   DoG_Keypts size : 75x75
Octave : 2   Scale : 3   DoG_Keypts size : 75x75
Octave : 2   Scale : 4   DoG_Keypts size : 75x75
Octave : 3   Scale : 0   DoG_Keypts size : 38x38
Octave : 3   Scale : 1   DoG_Keypts size : 38x38
Octave : 3   Scale : 2   DoG_Keypts size : 38x38
Octave : 3   Scale : 3   DoG_Keypts size : 38x38
Octave : 3   Scale : 4   DoG_Keypts size : 38x38
```

Figure 9: Scale Space Creation Console Output

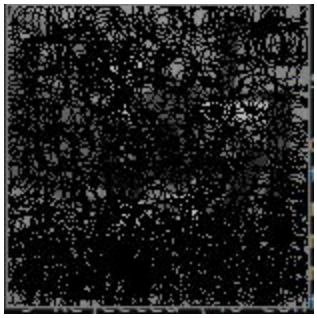

Figure 10: Scale-space Extrema Detection Output

```
Rejecting keypoints with low contrast or on edges...

Octave: 1 Scale: 1 Keypoints: 19 Rejected (205 contrast, 38 edge): 243
Octave: 1 Scale: 2 Keypoints: 1 Rejected (26 contrast, 2 edge): 28
Octave: 1 Scale: 3 Keypoints: 0 Rejected (4 contrast, 0 edge): 4
Octave: 1 Scale: 4 Keypoints: 82 Rejected (51 contrast, 99 edge): 150
Octave: 1 Scale: 5 Keypoints: 35 Rejected (14 contrast, 25 edge): 39
Octave: 2 Scale: 1 Keypoints: 9 Rejected (40 contrast, 13 edge): 53
Octave: 2 Scale: 2 Keypoints: 0 Rejected (5 contrast, 1 edge): 6
Octave: 2 Scale: 3 Keypoints: 0 Rejected (0 contrast, 0 edge): 0
Octave: 2 Scale: 4 Keypoints: 15 Rejected (14 contrast, 22 edge): 36
Octave: 2 Scale: 5 Keypoints: 9 Rejected (6 contrast, 6 edge): 12
Octave: 3 Scale: 1 Keypoints: 1 Rejected (7 contrast, 3 edge): 10
Octave: 3 Scale: 2 Keypoints: 0 Rejected (0 contrast, 0 edge): 0
Octave: 3 Scale: 3 Keypoints: 0 Rejected (0 contrast, 0 edge): 0
Octave: 3 Scale: 4 Keypoints: 7 Rejected (4 contrast, 6 edge): 10
Octave: 3 Scale: 5 Keypoints: 2 Rejected (4 contrast, 4 edge): 8
Octave: 4 Scale: 1 Keypoints: 0 Rejected (1 contrast, 0 edge): 1
Octave: 4 Scale: 2 Keypoints: 0 Rejected (0 contrast, 0 edge): 0
Octave: 4 Scale: 3 Keypoints: 0 Rejected (0 contrast, 0 edge): 0
Octave: 4 Scale: 4 Keypoints: 0 Rejected (2 contrast, 0 edge): 2
Octave: 4 Scale: 5 Keypoints: 1 Rejected (1 contrast, 0 edge): 1
```

Figure 11: Key Point Localization Console Output

Figure 12: Key Point Localization Output

```
Location[20][6]: Magnitude =  Orientation =
Location[20][7]: Magnitude =  Orientation =
Location[20][8]: Magnitude = ¿Orientation = «
Location[20][9]: Magnitude = ⌠Orientation = │
Location[20][10]: Magnitude = ùOrientation = ù
Location[20][11]: Magnitude = ¢Orientation = s
Location[20][12]: Magnitude = wOrientation = n
Location[20][13]: Magnitude = πOrientation = l
Location[20][14]: Magnitude = ▯Orientation = ▯
Location[20][15]: Magnitude = @Orientation = └
Location[20][16]: Magnitude = ¿Orientation = «
Location[20][17]: Magnitude = ⌠Orientation = │
Location[20][18]: Magnitude = ùOrientation = ù
Location[20][19]: Magnitude = ¢Orientation = s
Location[20][20]: Magnitude = wOrientation = n
Location[20][21]: Magnitude = πOrientation = l
Location[20][22]: Magnitude = ▯Orientation = ▯
Location[20][23]: Magnitude = @Orientation = └
Location[20][24]: Magnitude = ═Orientation = ╥
Location[20][25]: Magnitude = ;Orientation = !
Location[20][26]: Magnitude = ▯Orientation = 3
Location[20][27]: Magnitude = fOrientation = ▯
Location[20][28]: Magnitude = ₧Orientation = ▏
Location[20][29]: Magnitude = áOrientation = ┘
Location[20][30]: Magnitude = ▯Orientation = ▯
Location[20][31]: Magnitude = @Orientation = └
Location[20][32]: Magnitude = ┘Orientation = ╥
Location[20][33]: Magnitude = lOrientation = !
Location[20][34]: Magnitude = ▀Orientation = 3
Location[20][35]: Magnitude = ╞Orientation = ▯
```

Figure 13: Orientation Assignment Output

2.3 Discussion

Steps 1 and 2 of my SIFT implementation function as intended. Step 1 performs Scale-space extremum detection and step 2 performs key point localization, as described in [1]. The details of what occurred during these stages in the images above and make sense. Step 3, Orientation Assignment works as well, assigning magnitude and orientations to each key point. I could not figure out how to output the results of step 3. The arrowedLine() class in C++ takes as parameters the locations of the two endpoints of the arrowed line in the image. I could not figure out how to use calculate the second endpoint using the orientation and magnitude of the vector. I probably would have been able to do this given more time.

In the last step, I would have created a high dimensional vector storing the x and y location, the orientation, and magnitude of each key point. I would have used the method described in source [1] to assign a bin to each image gradient:

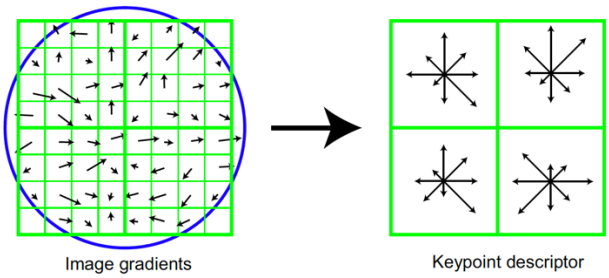

Figure 14: Key Point Descriptor Creation

This would have been able to be used in a nearest-neighbor algorithm that checks if two images are of the same subject.

2.4 Side Note

I took the GRE exam on Sunday, November 10, 2019. I was studying most of the week before and did not dedicate as much time to this project as I would have liked to. I still was able to work on this project for over 50 hours in total, and still only got this far. I just wanted to explain part of the reason why I did not entirely complete this assignment.

References

[1] Lowe, David. (2004). Distinctive Image Features from Scale-Invariant Keypoints. International Journal of Computer Vision. 60. 91-. 10.1023/B:VISI.0000029664.99615.94.
[2] Rani, Ritu & Grewal, Surender & Indiwar,. (2013). Implementation of SIFT in Various Applications. International Journal of Engineering Research and Development. 4. 59-64.
[3] phoenix16, "phoenix16/SIFT," *GitHub*, 01-Feb-2018. [Online]. Available: https://github.com/phoenix16/SIFT/blob/master/OpenCV/sift.cpp. [Accessed: 13-Nov-2019].