On Feature Encoding for Binary Descriptors
Master Thesis

Institution: Swiss Federal Institute of Technology
Sternwartstrasse 7, CH-8092 Zurich, Switzerland

Author: Michael Gygli, Student Master IFI
Advisor: Xavier Boix and Gemma Roig
Tutor: Dr. Marc Antonini
Supervisor: Prof. Dr. Luc van Gool

September 7, 2012
Abstract

Feature description is crucial in many Computer Vision tasks. The most widespread descriptor is SIFT (Scale-invariant feature transform) which has been published in 1999 and still ranks among the best performing descriptors available. However its performance comes at a cost: The resulting description vector has floating point values and is high dimensional, which results in a considerable need in memory and a high computation time, especially when comparing the features. This makes it unsuitable for real-time or large-scale applications.

This master thesis presents a novel method for calculating local features. It can be seen as a generalization of currently used descriptors (SIFT, BRIEF) and is highly parametrizable. It is based on a formulation developed by Boix et al.[4]. The focus of this work lay mainly on improving the encoding step of the existing descriptors. As a result the final descriptor is comparable to SIFT in some aspects, for example in the arrangement of the spatial bins. However, SIFT filters the input with x- and y-gradients and uses a histogram of oriented gradients as a codebook. On the other hand, we use a more general formulation for the image filters, which we encode with a codebook that differs from the one of SIFT. With our formulation it is possible to use any number of filters and arbitrary codebooks. For the default configuration of our method, we propose to use four different gradients and a codebook, for which assignment can be done via a fast sorting algorithm. As a last step our method binarizes the resulting vector, using sparse quantization, which results in a compact binary descriptor. Experiments on the datasets of Brown and Mikolajczyk show that the descriptor outperforms SIFT and achieves state-of-the-art performance. Furthermore this thesis compares the speed of the descriptor in its different process steps and demonstrates the usefulness of it in large-scale matching. Finally, it shows the descriptors potential in 3D reconstruction with Structure from Motion.
Acknowledgments

Mainly I want to thank my main advisor Xavier Boix. He introduced me to the world of scientific research and helped me getting familiar with the field of my thesis. He always took the time to help me and to give me new valuable input. I would also like to thank Gemma Roig, my co advisor, for her inputs.

Thank goes also to the students which wrote their thesis in the same Lab, as they were of valuable help for small, technical problems. Last, but not least special thanks goes to Théo Papadopoulos, Marc Antonini and especially Frédéric Precioso for their inspiring courses and for awaking my interest in the field of feature detection and matching and 3D reconstruction, which lead to my master thesis.
Contents

1 Introduction ................................................................. 1
   1.1 Thesis Organization .................................................. 3

2 Related Work ............................................................... 5

3 Method ........................................................................... 7
   3.1 Feature Extraction ....................................................... 7
      3.1.1 Filtering ............................................................. 7
      3.1.2 Codebook assignment ............................................. 8
      3.1.3 Pooling ............................................................. 10
      3.1.4 Binary Sparse Quantization ...................................... 11
   3.2 Matching Strategies .................................................... 12
   3.3 Learning filters .......................................................... 12
      3.3.1 Learning based on performance ................................ 12
      3.3.2 Learning based on the Sparsity criterion ...................... 13
      3.3.3 Implementation details .......................................... 13
   3.4 Use for BRIEF-like descriptors ....................................... 14
      3.4.1 Filtering ............................................................. 14
      3.4.2 Codebook assignment ............................................. 15
      3.4.3 Pooling ............................................................. 15

4 Experiments and Results ..................................................... 17
   4.1 Filtering .................................................................... 18
      4.1.1 Amount of Filters ............................................... 19
      4.1.2 Choice of the filter region ..................................... 20
      4.1.3 Learning filters .................................................. 20
   4.2 Codebook .................................................................. 22
   4.3 Binary Sparse Quantization .......................................... 24
   4.4 Influence of KNN and Beta .......................................... 24
   4.5 Pooling .................................................................... 25
   4.6 Smoothing .................................................................. 25
   4.7 Variations of BRIEF ................................................... 26
   4.8 Comparison to the state-of-the-art ................................ 27
      4.8.1 Multi-view Stereo Correspondence Dataset ................. 32
      4.8.2 Mikolajczyk dataset .............................................. 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.8.3</td>
<td>Speed comparison</td>
<td>35</td>
</tr>
<tr>
<td>4.8.4</td>
<td>SIFT parametrization</td>
<td>35</td>
</tr>
<tr>
<td>4.9</td>
<td>3D Reconstruction</td>
<td>36</td>
</tr>
<tr>
<td>4.9.1</td>
<td>Feature extraction</td>
<td>36</td>
</tr>
<tr>
<td>4.9.2</td>
<td>Multiview correspondences</td>
<td>36</td>
</tr>
<tr>
<td>4.9.3</td>
<td>Initialization</td>
<td>37</td>
</tr>
<tr>
<td>4.9.4</td>
<td>Bundle Adjustment</td>
<td>39</td>
</tr>
</tbody>
</table>

5 Conclusion 43
# List of Figures

1.1 Keypoints detected with SIFT .................................................. 1  
1.2 Keypoints matched using BG-SIFT ............................................. 2  
3.1 Histogram of Gradients .......................................................... 8  
3.2 Different Spatial Pooling strategies ........................................... 11  
3.3 Some candidate filters for the learning algorithm ......................... 14  
3.4 Brief-like filters. (a) Filters of BRIEF (b) Filters of G-BRIEF (c) Subset of the pairwise filters 15  
4.1 Dimensions dependent on the number of filters .......................... 19  
4.2 BG-SIFT dense and less dense. (a) Error rate at 90% (b) Speed (fixed-scale) .......................... 20  
4.3 Performance and speed (fixed-scale) of dense BG-SIFT and less dense BG-SIFT (delta=2) .......... 21  
4.4 SIFT dense and less dense. (a) Performance on Liberty (b) Performance on Yosemite ................. 22  
4.5 Performance of BG-SIFT and less dense BG-SIFT (delta=2) on the Mikolajczyk dataset ........ 23  
4.6 Filters for dense filtering. (a) Initial filters (b) learned filters (c) filters that generally offer a well performance. Red: Current pixel Blue: Positive part Green: Negative part .......... 24  
4.7 Performance on Yosemite with the learned filters ........................ 25  
4.8 Speed of BG-SIFT with distance calculation or sorting (scale-aware) ....................................... 25  
4.9 Performance of 3 different random codebooks compared to a $\mathbb{R}^n_k$ codebook .................. 25  
4.10 Performance for partial codebooks. (a) Liberty (b) Yosemite .................. 26  
4.11 Performance dependent on k for the Sparse Quantization. (a) For SIFT (b) For G-SIFT .............. 27  
4.12 Performance of (Binary)-SIFT and (Binary)G-SIFT on the Yosemite dataset ...................... 28  
4.13 Performance with and without binarization on images from the Mikolajczyk dataset .............. 28  
4.14 Beta optimization of BG-SIFT with the 90% error rate .................... 29  
4.15 Performance of BG-SIFT with KNN=1 and KNN=2 ..................................... 29  
4.16 Average Pooling vs Max Pooling .............................................. 30  
4.17 Influence of pre-smoothing on yosemite .................................... 31  
4.18 Performance of BG-SIFT with different smoothing strategies ........... 31  
4.19 Performance of G-BRIEF. (a) Performance on Liberty (b) Performance on Yosemite ............... 32  
4.20 Speed and filters of G-BRIEF. (a) Speed per 512 descriptors (b) Four centered filters ............. 33  
4.21 ROC-curves for the performance with the Brown test setup .................. 34  
4.22 Performance BRIEF test setup on Liberty .................................. 35  
4.23 Performance on image sets with scale and rotation ........................ 36  
4.24 Performance comparison on the Mikolajczyk dataset ....................... 37  
4.25 Construction speed of the different descriptor parameters (scale-aware) ....................... 38  
4.26 Speed for description (scale-aware) and matching .......................... 40
LIST OF FIGURES

4.27 Performance comparison of Vedaldi SIFT to our implementation of SIFT . . . . . . . . 40
4.28 3D reconstructions of household objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
## List of Tables

4.1 Speed of BG-SIFT compared to SIFT ................................................. 19  
4.2 Description speed of BG-SIFT (scale-aware) and less dense BG-SIFT (delta=2). .......... 23  
4.3 False positive rate at 95% recall for KNN=1 and KNN=2 .................................. 26  
4.4 Description and matching speed (scale-aware) for different KNN .......................... 28  
4.5 False positive rate at 95% recall on Browns datasets ........................................... 33  
4.6 Performance on the Liberty dataset ......................................................... 34  
4.7 Speed for description (scale-aware) and matching .............................................. 38
Chapter 1

Introduction

The goal of this thesis is to analyze the influence of the encoding phase in keypoint description. In order to do so, we propose to use a more elaborate encoding than existing descriptors such as SIFT[13].

Keypoints are characteristic points within an image. Such a point can be a corner or any other point which has strong differences in intensity compared to its neighbors. Ideally a keypoint detection algorithm is robust to noise, rotation, scaling, viewpoint and light changes, i.e. it obtains the same keypoints on an image or object which suffers under such transformations. A good and widely used detector can be for example found in SIFT (Scale-invariant feature transform)[13]. Keypoints obtained with this detector are shown in Figure 1.1.

In this work we focus on the descriptors of such keypoints. The goal of the keypoint description is to find a representative vector which describes the keypoint in a distinctive but robust way. With these feature vectors it is then possible to create correspondence between the keypoints of different images. An example of such correspondences or matches is shown in Figure 1.2. Good feature descriptors are, as the detectors, robust to light changes, viewpoint changes, noise, blur, rotation and scaling. This means that a keypoint suffering under such transformations should always lead to a similar description vector.

To match keypoints from different images, the keypoints with the smallest distance of in their descriptor vectors are considered to be corresponding points (i.e. matches). In the case of the SIFT descriptor [13] the

Figure 1.1: Keypoints detected with SIFT
CHAPTER 1. INTRODUCTION

Figure 1.2: Keypoints matched using BG-SIFT
(correct: green, wrong: red)

feature vector contains floating point values and the distance is calculated using the Euclidean distance. In the case of BRIEF[6] this vector is binary.

Using binary vectors allows for a huge improvement in speed for the matching phase, as the distance can be calculated using the Hamming distance. On current CPUs this can be done using the Population Count instruction (POPCNT) which accelerates matching by orders of magnitude. While BRIEF is very fast in both, description and matching, it has a big trade-off in performance.

This thesis presents a descriptor that fills the gap between fast and high performing descriptors. It proposes a binary descriptor that performs better than SIFT, while being significantly faster, especially in the matching phase. The newly developed descriptor, called BG-SIFT, can be described as a generalization of SIFT which is binarized using Sparse Quantization, as proposed by Boix et al.[4]. It is similar to SIFT in some aspects, but has a more elaborate encoding, which results in an improved performance and the possibility to binarize the descriptor without losing matching accuracy. The binary representation allows for fast descriptor matching, where BG-SIFT is four times faster than SIFT. The descriptors theory is explained in Chapter 3. It was then tested extensively on the widely used, publicly available datasets of Matthew Brown. Furthermore its potential was shown in matching on the test sets of Mikolajczyk (Graffiti etc.) and used in an application of 3D reconstruction.
1.1 Thesis Organization

- **Related Work** This section puts the BG-SIFT descriptor into context with the current state-of-the-art descriptors.

- **Method** Here the theory and methods behind the descriptor are explained.

- **Experiments and Results** The achieved performance and speed of BG-SIFT are analyzed and compared to existing descriptors in this section. Furthermore an application in 3D reconstruction is shown.

- **Conclusion** This section discusses the achieved results and glances and further possible improvements and applications.
Chapter 2

Related Work

Keypoint description is a very active field of research. In 1999 Lowe presented SIFT [13], which is the most used descriptor and still ranks among the best performing ones. SIFT combines a Keypoint detector and a descriptor. The detector is based on a Harris corner detection [8] which was improved to provide scale and rotation of the Keypoint. While detection is of high importance for any feature description, this thesis lays focus on the description of keypoints. In SIFT the descriptor makes use of a Histogram of oriented Gradients (HoG). Such a histogram is built by calculating the gradients in the x- and y-direction and then a calculation of the direction of the gradient. Each gradient direction is then assigned its closest orientation in the histogram (with steps of 45°).

Our descriptor has a similar approach, but it generalizes the use of gradients. It uses gradients in multiple directions (four with the default parametrization). As a result it is no longer possible to work with the gradient direction. Instead a more general codebook of variable size is used. A filter response is assigned to its closest elements in the codebook.

While SIFT is still the state-of-the-art in feature matching, it has a high computational time. It is generally too slow for any real-time application. As a consequence the focus of research laid mostly on developing fast descriptors in the last years. Such a descriptor is SURF by Bay et al. [3]. SURF offers a good compromise in trade-off between performance and speed which makes it suitable for real-time applications such as Augmented Reality. Even faster than SURF are binary descriptors, which came up recently. They generally offer an outstanding speed boost, which is due to the fact, that binary vectors can be compared orders of magnitude faster than floating-point vectors. The most used among the binary descriptors are BRIEF [6] and ORB [18], which is based on BRIEF, but comes along with several additions such as a detector (based on FAST [17]) and an orientation estimation. Others binary descriptors are CARD [2] and BRISK [11]. Just now Trzcinski and Lepetit published a new paper proposing an new, improved version of BRIEF (called D-BRIEF)[20], which uses learned projections to get a low dimensional binary descriptor with good performance. This descriptor exceed the existing binary descriptors in both, speed and performance.

While all these binary descriptors are orders of magnitude faster than SIFT, their performance is always much lower than the one from SIFT (See Table 4.6 for a comparison). Other recent papers propose feature learning ([19],[5]) to increase the performance and accomplish to outperform SIFT. Thereby they accept a high computational cost. This gap is where we lay the focus of our work: Learning a fast binary descriptor which achieves state-of-the-art performance. Our descriptor is similar to SIFT, but has an improved encoding phase. Furthermore the descriptor is binarized using Sparse Quantization as proposed by Boix et al.[4]. The binary representation of our descriptor results in a matching time that is four times lower than the one of SIFT, which makes it especially useful in applications with a high amount of descriptors to match.
Such applications might be image retrieval or large-scale Structure from Motion, such as the “Building Rome in a Day” project [1]. In this project Agarwala et al. did a 3D reconstruction of Rome based on 100,000 tourist photographs found on photo-sharing websites from which they extracted 5,000,000,000 SIFT-features, which need to be matched.
Chapter 3

Method

In this Chapter we describe the method of our descriptor. This is done in section 3.1. Thereby we often use our best performing parametrization as an example, but the described formulation can be used for any parametrization. This is shown in section 3.4, where we use the formulation to create a descriptor similar to BRIEF.

3.1 Feature Extraction

Let $K$ be the set of keypoints in an image, and we denote as $f_i \in \mathbb{R}^q$ the feature vector of the keypoint, where $i \in K$. This feature vector is calculated using a patch of fixed size of $S \times S$ pixels, centered at the keypoints coordinates. In order to calculate a feature vector, the following operations are carried out:

Filtering. Calculation of the filter responses on the patch.

Codebook assignment. In this step, the filter responses are assigned to their $k$-nearest neighbors in a codebook. The filter response then contributes to these $k$ elements in the pooling phase.

Pooling. This step summarizes the filter responses into a vector. There exist various pooling strategies, such as max pooling, average pooling and concatenation. For our descriptor we test all of those, depending on the configuration. This techniques and their use is further explained in section 3.1.3.

Binary Sparse Quantization. Sparse Quantization, as defined by Boix et al. [4], is a quantization of a real-valued vector to its nearest neighbor in a set of $k$-sparse vectors. We use a set of binary vectors. A pooled vector of dimension $n$ is therefore quantized to a binary vector of dimension $n$, with a number of non-zero dimensions equal $k$.

Note that in SIFT, there is an additional step of normalization. In this step, the descriptors length is normalized to one. Since our method binarizes the descriptor, this step is unnecessary (see section 3.1.4).

3.1.1 Filtering

In filtering, a set of filters is applied pixel by pixel on an area of $S \times S$ pixels around a keypoint $K_i = (x_k, y_k)$, i.e. we apply it for:
SIFT has two filters, the gradients in the x- and y direction:

\[
    g_x(x, y) = I(x + 1, y) - I(x - 1, y) \\
    g_y(x, y) = I(x, y + 1) - I(x, y - 1)
\]

they are applied on the above area. \( I(x, y) \) is the image intensity at point \((x, y)\).

We propose the use of \( n \) filters. As shown in the experiment section, four filters offer a convincing improvement in performance while being reasonably fast. Therefore we propose to use four filters like the following, for example:

\[
    g_1(x, y) = I(x + 1 + \Delta, y) - I(x + 1, y) \\
    g_2(x, y) = I(x - 1 - \Delta, y) - I(x - 1, y) \\
    g_3(x, y) = I(x, y + 1 + \Delta) - I(x, y + 1) \\
    g_4(x, y) = I(x, y - 1 - \Delta) - I(x, y - 1)
\]

In our final version of the descriptor, we use \( \Delta = 4 \), while SIFT has \( \Delta = 2 \). These filters are shown in Figure 4.6 (c).

### 3.1.2 Codebook assignment

In this step, the filter responses are assigned to an arbitrary codebook. Here we present a uniform codebook, as this is similar to SIFT and it allows for fast assignment using a sorting algorithm.

In SIFT, this codebook is a set of oriented gradients, as shown in Figure 3.1. With the x- and y-gradients, the angle of orientation is calculated. Dependent on this angle, the filter response is the coded with the two-nearest codebook elements, which are the two closest angles.

While a codebook entry can be seen as a certain angle, it can also be seen as a vector of size \( n \) with \( k \) activated dimensions (\( n = 2 \) in the case of SIFT). This is equivalent to choosing \( k \) out of \( n \), which is known
as the binomial coefficient. We denote \( B_n^k \) as the set of \( k \)-sparse vectors of dimension \( n \). The codebook of SIFT then consists of all combinations of \( 2 \)-dimensional vectors with \( k \) dimensions activated at a time, i.e. \( \bigcup_{k \in 1..2} B_k^2 \). Since negative gradients are possible (\( g_x(x, y) < 0 \) or \( g_y(x, y) < 0 \)), the codebook further contains all combinations with negative activation. An example: An angle of \( 0^\circ \) lies on the x-axis, i.e. only the \( g_x(x, y) \) is (positively) activated. Therefore this angle is equivalent to the codebook element \( \begin{pmatrix} 1 \\ 0 \end{pmatrix} \). \( 135^\circ \) is coded by \( \begin{pmatrix} -1 \\ 1 \end{pmatrix} \) and so on (see Figure 3.1).

The number of codebook elements, defined as above can be calculated as

\[
C = \sum_{k=1}^{n} 2^k \cdot |B_k^n| = \sum_{k=1}^{n} 2^k \cdot {n \choose k}
\]

This definition of a codebook can easily be used for any number of filters. For the case of four filters, the codebook elements range from \( (1, 0, 0, 0) \) to \( (1, 1, 1, 1) \) or \( (-1, -1, -1, -1) \), respectively. The codebook elements are: \( \bigcup_{k \in 1..4} B_k^4 \) (and all the combinations with negative activation). Note that we normalize all the vectors of the codebook to length one, using the \( l_2 \)-norm.

A filter response vector

\[
\begin{pmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{pmatrix}
\]

is assigned to the \( k \)-nearest neighbor (KNN) elements in the codebook. This is done using an exponential weight \( \alpha_i^* \) which is defined as follows:

\[
\alpha_i^* = \begin{cases} 
\exp(-\beta d(g_{1..4}, b_i)) & \text{if } i \in k\text{-NN}(x, B) \\
0 & \text{otherwise}
\end{cases}
\]

where \( k\text{-NN} \) is the set that indicates which \( k \) codebook entries are the closest to \( g_{1..4} \), \( \beta \) is a learned constant and \( d(a, b) \) is the distance to the codebook element. This weight function is used as in [4]. The sum of the \( k \) weights is normalized to one.

In the case of \( KNN=1 \), this weight is one for the nearest neighbors and zero for the others, which makes the exponential distance calculation unnecessary.

The above described uniform codebook allows doing codebook assignment with the use of an approach based on sorting. In a first step, the filter response is sorted by its absolute value, in a decreasing order. Then, the closest elements to \( B_k^4 \) are the codebook elements using the \( k \) highest values. If we have for example: \( (g_3 \ g_1 \ g_2 \ g_4)^t \) we get the closest \( B_1^4 \) equal \( (0, 0, 1, 0) \), \( B_2^4 \) equal \( (1, 0, 1, 0) \) and so on. This is in the case that all values \( g_{1..4} > 0 \). Otherwise, if, for example, \( g_3 < 0 \), \( B_1^4 \) equals \( (0, 0, -1, 0) \) and so on. In order to find the nearest neighbor among them, the four distances for the filter response to \( B_1^4 \) to \( B_4^4 \) are calculated and the \( k \) closest ones are selected.

Therefore the codebook assignment can be done with two sortings of \( n \) elements and \( n \) distance calculations, where \( n \) is the number of filters. The naive approach needs to calculate the distances to all the codebook elements, which increase exponentially with \( n \) (See Figure 4.1). Algorithm 1 shows the pseudo code for this optimized codebook assignment. In Figure 4.8 of the Experiment section, we compare the speed of the two approaches.
CHAPTER 3. METHOD

Algorithm 1: Fast codebook assignment based on sorting

<table>
<thead>
<tr>
<th>Input: filter responses $g[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: $k$-nearest codebook elements $c[k]$</td>
</tr>
<tr>
<td>/* 1) Sort the filter responses $g$ and store the the indices sorted of the sorting in $idx$ (sorted by absolute value and in a descending order) */</td>
</tr>
<tr>
<td>$\text{Sort}(g, idx)$</td>
</tr>
<tr>
<td>$c_{tmp}[n]$ /* Temporary codebook elements */</td>
</tr>
<tr>
<td>for $k_{tmp} = 1$ to $n$ do</td>
</tr>
<tr>
<td>/* Construct the codebook element */</td>
</tr>
<tr>
<td>for $i = 1$ to $k_{tmp}$ do</td>
</tr>
<tr>
<td>if $g[idx[i]] &gt; 0$ then</td>
</tr>
<tr>
<td>$c_{tmp}[k_{tmp}][idx[i]] = 1$</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>$c_{tmp}[k_{tmp}][idx[i]] = -1$</td>
</tr>
<tr>
<td>$\text{dist}[n]$ /* distances to codebook elements */</td>
</tr>
<tr>
<td>/* 2) calculate the $n$ distances */</td>
</tr>
<tr>
<td>for $k_{tmp} = 1$ to $n$ do</td>
</tr>
<tr>
<td>$\text{dist}[k_{tmp}] = \text{distance}(g, c_{tmp}[k_{tmp}])$</td>
</tr>
<tr>
<td>/* 3) Sort the distances and store the indices in $idx$ (sorted in an ascending order) */</td>
</tr>
<tr>
<td>$\text{Sort}($dist, $idx)$</td>
</tr>
<tr>
<td>/* Return the $k$ nearest codebook elements */</td>
</tr>
<tr>
<td>return $c[idx[1]..idx[k]]$</td>
</tr>
</tbody>
</table>

3.1.3 Pooling

The output of the above steps is, for every pixel $(x, y)$ of the patch, the nearest neighbor(s) in the codebook and the weighted magnitude for the filter response. There exist the following pooling strategies to pool these values into a vector:

**Average Pooling.** In this case, the magnitude or activations of a codebook element are averaged. This results in a vector with the average activation (magnitude) per codebook element.

**Max Pooling.** This pooling strategy builds a vector with the maximal activation per codebook element.

**Concatenation.** In concatenation, values are not actually pooled. Instead, all the filter responses are concatenated. This results in a very high-dimensional vector, containing all the values of the before matrix. Concatenation is used for BRIEF, since there are only 256 tests (by default), which leads to a 256-dimensional vector. In the case of our descriptor, this is not practicable, since the filters are applied densely, for every pixel of the patch.

**Spatial Pooling.** Instead of pooling all the filter responses together, spatial pooling takes the spatial distribution of a pixel into account. This can be done with various techniques. SIFT groups the results of the codebook assignment phase into $4 \times 4$ spatial regions (see Figure 3.2). A filter response in position
Figure 3.2: Different Spatial Pooling strategies

(a) Spatial Pooling of SIFT (taken from [13])  (b) Spatial Pooling of CARD (taken from [2])

$(x, y)$ contributes to the four closest bins. Furthermore the weight of this contribution is dependent on the distance to the keypoint. For the weighting, a Gaussian function with $\sigma$ equal half of the descriptors scale is used. As it is dependent on the scale, it needs to be computed for every pixel, for every keypoint, which is computationally expensive, as we show in 4.25. Our descriptor uses the same pooling, although there are more natural spatial poolings, such as the one from CARD [2], which is shown in Figure 3.2. Also, recent papers ([5], [20], [19]) are dedicated to learn a more sophisticated spatial pooling and have shown the potential over a conventional spatial pooling.

We compared average pooling against max pooling in section 4.5. The two strategies perform similar, with average pooling showing a slight advantage over Max pooling.

SIFT uses $4 \times 4$ spatial bins, which results in a description vector that has dimensions $d = 16 \times |C|$ where $|C|$ is the size of the codebook. We use the same spatial pooling for our descriptor. In the case of four filters, we have $|C| = 80$, which leads to 1280 dimensions. While this descriptor has shown a high performance, it is an order of magnitude larger than SIFT, which results in a high need for storage and a matching time that is also an order of magnitude longer. This problem can be solved by quantizing the floating point vector into a binary vector as described in the following section.

### 3.1.4 Binary Sparse Quantization

Sparse Quantization, as proposed by Boix et al. [4], is a quantization of a real-valued vector to its nearest neighbor in a set of $k$-sparse vectors. This is very similar to what we do for the codebook assignment (see section 3.1.2). A pooled vector of dimension $n$ is therefore quantized to a binary vector of dimension $n$, with a number of non-zero dimensions equal $k$. In order to do so, we use again a sorting algorithm to sort
Chapter 3. Method

the vectors values in a descending order while retaining the indices of these values. The binary descriptor can then simply be created by setting the indices of the $k$ highest values to one and the rest to zero.

Empirically we tested the performance of our descriptor for various values of $k$ (Figure 4.11) and found that values with $k \approx n/5$ give the best results. This is in the case of four gradient filters, a codebook of 80 elements and 16 spatial bins. Note that the value of $k$ is dependent on the sparsity and dimensionality of the real-valued descriptor. Section 4.3 shows the optimal values we found for binarized SIFT and BG-SIFT. When using complex filter a more sparse vector can be learned, which then has a lower optimal value for $k$. See section 3.3.2 for the learning of complex filters.

3.2 Matching Strategies

In order to match the features of different images, there are multiple strategies possible. In the course of this thesis we used three, which are the following:

**Match Left-Right.** This matches features from a first set to a second set (nearest-neighbor matching) and vice versa. It then applies a consistency check, only accepting correspondences as a match, that are consistent in their nearest-neighbor, i.e. a match is only made, if feature $b_{1i}$ has $b_{2j}$ as its nearest-neighbor and $b_{2j}$ has $b_{1i}$.

**Match First to Second.** This strategy matches every feature from a first set to the nearest-neighbor feature of the second set (as used in BRIEF[6]).

**Ratio matching.** Ratio matching calculates to 2-nearest-neighbors. A match is then created when the ratio between the first-nearest-neighbor to the second-nearest-neighbor is below a threshold:

$$\frac{\text{first-nn}}{\text{second-nn}} < t$$

with threshold $t$. If this ratio is small, the distance to the second nearest-neighbor is much bigger than to the nearest-neighbor, which indicates a reliable match. The $t$ is given as a parameter and is usually $\approx 0.8$, as used in [7] to build robust multiview correspondences.

3.3 Learning filters

In order to find optimal filters, we tested two approaches: In section 3.3.1 we learned the use of good gradients to optimize the performance. Then, in section 3.3.2, we extracted patches from various images, normalized them to have mean zero and used them as filters for the filtering phase, instead of simple gradients. Here, we optimized the Sparsity, instead of optimizing the performance.

3.3.1 Learning based on performance

This learning was implemented as a simple greedy algorithm. For each iteration, four filters are generated, as a difference of two random pixels. The pixels are chosen in the range of $[-5; 5]$. With these filters we calculated the features on the training images and evaluated the matching performance. If the set of filters leads to better performance, it was kept. This was done a few hundred iterations, and in the end the best performing gradient filters were applied on the test set.

See section 4.1.3 for results.
3.3.2 Learning based on the Sparsity criterion

We further used learning for complex filters. Complex, in this context, means filters that take more pixels into account than a simple gradients (i.e. a difference between two pixel intensities). The idea was to learn filters, that yield a sparse representation of the real-valued descriptor. Various papers were published about this topic. Our work is thereby largely based on the ideas presented by Andrew Ng and his team[15]. Experiments have shown that the primary visual cortex reacts on visual patterns with a sparse activation of neurons. This findings lead to the learning of sparse filters, as sparsity has proven to offers a good criterion for the performance of a set of filters[16]. The learning algorithm proposed by Ng et al. exploits the same idea, proposing a learning algorithm, that enforces Population Sparsity and High Dispersal (and therefore lifetime sparsity, as stated in [15]).

**Population Sparsity.** (Sparse features per example) Every descriptors should have a low number of active (non-zero) elements.

**High Dispersal.** (Uniform activity distribution) In order to prevent degenerate situation, where some elements are always activated, while others are never activated, it is necessary to enforce Uniform activity distribution. So, to enforce that every elements is activated around the same amount, overall the features in the training set.

**Lifetime Sparsity.** (Sparse features across examples) This means an element should be activated only in a few features of the training set.

Closely following the algorithm proposed by Ng, we minimize the following function $s$:

$$s(g_1, g_2, g_3, g_4) = \sum_{i=1}^{M} \left\| \hat{f}^{(i)}(g_1, g_2, g_3, g_4) \right\|_1$$

where $g_1, g_2, g_3, g_4$ are the candidate filters and $\hat{f}^{(i)}(g_1, g_2, g_3, g_4)$ is the feature of keypoint $i \in \mathcal{K}$, which has its elements normalized by their lifetime activation:

$$\hat{f}_j(g_1, g_2, g_3, g_4) = \frac{f_j(g_1, g_2, g_3, g_4)}{\|f_j(g_1, g_2, g_3, g_4)\|_2}$$

The candidate filters are chosen as described in the next section. The algorithm is implemented as a greedy optimization (see 3.3.3 for details). Other than Ng, we tried to learn filters as an input for the codebook assignment step. Instead of simple gradients, complex filter, should be learned and the filter responses than assigned to a codebook, as before. See section 4.1.3 for results.

**Candidate filters**

In order to obtain candidate filters, over 2000 patches are extracted from a the Mikolajczyk dataset. These patches are normalized to have zero mean. Then, they are clustered to 400 filters using k-means. These are finally used as candidate filters in the learning algorithm. A few examples of those input filters can be seen in Figure 3.3.

3.3.3 Implementation details

The learning algorithms are implemented as a greedy algorithms. They randomly choose a set of four filters and calculate the resulting features over a set of training images. For the above sparsity learning the
overall sparsity is evaluated, as described before. If the feature is more sparse than currently best filters, the filters are accepted as the currently best ones and the procedure is reiterated. For performance learning, the performance on the training set is evaluated instead. This algorithm is repeated for a few hundred iterations, after which it converges. In order to implement the algorithm in an efficient manner, multithreading was used. This is implemented using the Boost thread library. As calculation of complex filter responses is computationally expensive, the filters response of all the candidate filters are precomputed for all the training images, in the case of learning complex filters. This results a reduction of the time for an iterations by the factor of 6.

**Training data.** The training data is generated through applying synthetic transformations on a set of images. These transformations are: Blur, small angles of rotation and viewpoint changes.

### 3.4 Use for BRIEF-like descriptors

In this section we show how the above formulation can be used to create a descriptor such as BRIEF. Other than BRIEF, we use different filters, which have shown to bring better performance.

#### 3.4.1 Filtering

While SIFT has two filters, which are applied densely on the patch of the keypoint, BRIEF has a number of different filters (256 by default), which are then only applied once on the patch. These filters are chosen randomly over the size $S$ of the patch, with a Gaussian distribution with mean zero and $\sigma = S/5$.

We propose to use a different filtering. Instead of centering all the tests around the keypoint, we use more local filters. For this we use a $\sigma = S/8$. These filters are then applied at four different locations around the keypoint. Figure 3.4 shows an example of such filters.
3.4. Codebook assignment

Codebook assignment is very simple in BRIEF like descriptors. The codebook only consists of two elements: (1) and (−1). BRIEF combines the step of binarization in the step of codebook assignment. If a filter response is > 0, the corresponding dimension is coded with 1, otherwise with 0.

We tested using a more elaborate codebook, namely one that is similar to SIFT. For this we use pairwise filters. We generate random filters and for every filter we calculate the filter that’s perpendicular to it (see Figure 3.4 (c) ). With this pairs, we use a codebook of 8 elements, as in SIFT, and code the pairwise filter responses as described in section 3.1.2. For this step a KNN=2 is used with assignment counting, i.e. two elements of the codebook are set to one, the others to zero.

3.4.3 Pooling

As in BRIEF, no Pooling is done. All the results from the coding phase are concatenated into a vector. Note that through the use of four differently centered sets of tests, this descriptor has some notion of spatiality, even if there is no spatial pooling.

In section 4.7 we present the performance and speed of this fast descriptor and compare it to BRIEF.
CHAPTER 3. METHOD
Chapter 4

Experiments and Results

In this Chapter we test the influence of different parameters on the speed and performance of the descriptor. This is done with two datasets: The Multi-view Stereo Correspondence Datasets of Brown and the Mikolajczyk datasets.

The rest of this section is organized as follows: Sections 4.1 to 4.6 investigate the influence of the filters, codebook and other parameters on the performance and speed. Section 4.7 does performance tests with our version of BRIEF. Then, section 4.8 compares our final descriptors, which we name binary general SIFT (BG-SIFT), to the state-of-the-art. Finally, section 4.9 shows an application of BG-SIFT in 3D reconstruction.

Multi-view Stereo Correspondence Datasets. This datasets are widely used in papers like [6],[5] and [19]. They consist of corresponding patches sampled from 3D reconstructions of the Statue of Liberty (New York), Notre Dame (Paris) and Half Dome (Yosemite). Every dataset contains 400k patches and every patch has its real-world-point-id associated. The patches have a size of $64 \times 64$ pixels. Our descriptor takes a smaller area into account. We optimized the size of this area in section 4.1.2.

We use a test set of $10^4$ or $100^4$ pairs with 50% matching patches and 50% non-matching patches in all cases. For every pair, the distance is calculated. If the distance is below a threshold $t$, it is assumed to be a match. I.e. when $d(f_i, f_j) < t$. This threshold may be fix, when optimizing some parameters or varied to build the ROC-curve. This follows exactly the experimental setting of [5] which makes our results comparable to the ones given there and to other recent papers, such as [19]). Often we optimize the 90% error rate, which is the percent of incorrect matches, when 90% of the true matches are found. For all figures of this section the dataset and the size of the used subset are specified in the title.

The dataset is available on the website of Matthew Brown\(^1\).

Mikolajczyk dataset. For many performance tests we used the image test sets of Mikolajczyk and C. Schmid [14]. This dataset is available online\(^2\). The dataset contains images covering the standard transformations a descriptor should be invariant to. The Wall and Graffiti images cover viewpoint changes, Tree and Bike cover blur, JPEG compression artifacts and the Light set illumination changes. For all the test with this dataset we use keypoints detected with the Vedaldi implementation of SIFT. SIFT detects keypoints at a certain scale. In order to do so, it creates an image for every scale. To reduce computation time, we reused these images to calculate our descriptors upon, i.e. we calculate the descriptors on the exact same

---

\(^1\)http://www.cs.ubc.ca/~brown/patchdata/patchdata.html

\(^2\)http://www.robots.ox.ac.uk/~vgg/research/affine/
CHAPTER 4. EXPERIMENTS AND RESULTS

input as the SIFT descriptor does. The downside of this approach is, that the scaling is not optimized for BG-SIFT, which generally needs less smoothing than SIFT. Furthermore we use a patch size of 48 pixel to compute the descriptor on. In order to achieve the most independence possible from the detector, the keypoints are validated using the ground truth. After the detection phase every keypoint from the first image is projected on the second, using the homography given in the test set. A keypoint is then only kept, if it has a corresponding keypoint at this projected position (with a tolerance of 2 pixels, as this is the tolerance in the verification of a correct match). Equally the keypoints of the second image are only kept, if they are a projection of the first ones. With this procedure, we only match features that have a correct correspondence (but small shifts are possible). Furthermore this provides the descriptor algorithm with a scale and orientation computed from the detector, which are needed for the scale and rotation tests. Matching is done using a first-to-second matching, as described in section 3.2. For all upright tests we use the following parameters for the SIFT detector: threshold = 0.02, edgeThreshold = 10, first_octave = -1. For all scale and rotation tests we use: threshold = 0.001666, edgeThreshold = 150, first_octave = 0. See [21] for more information about the Vedaldi implementation of SIFT and its parameters.

Speed tests. For tests for the description times, the following setup was used: From an image \( \approx 2000 \) keypoints are extracted. These are then described with the respective descriptor. This is done 10 times. The final speed is calculated as an average of these 10 runs and given in milliseconds per 512 descriptors.

Please note that there are two possible uses of the BG-SIFT descriptor: scale-aware or fixed-scale. If the descriptor is scale-aware, the Gaussian weight needs to be computed, dependent on the scale of the descriptor (see 3.1.3). As this weight is exponential, the computation is time consuming. For some tasks, such as classification, descriptors are computed at a fixed scale. Therefore this Gaussian weight needs to be computed only once. This makes the description time much faster. In the speed tests, we use both of these configurations, dependent on the dataset we use. For the Brown dataset we use the descriptor with a fixed scale configuration and for the Mikolajczyk we use the scale-aware version. All speed tests specify which version was used. Notice that we give the times without pre-smoothing. The reason for this is, that BG-SIFT can be calculated on the pre-smoothed images created by the SIFT detector or it can be calculated on an unsmoothed patch. We therefore compare BG-SIFT to SIFT and BRIEF, not taking into account the smoothing in either case.

For the matching speed, random descriptors of the given descriptor length and type (float or bit) are matched. In matching, the smallest distance between two descriptors is searched. This means that every descriptor need to be compared to all the other descriptors. Therefore this takes quadratic time \( (O(n^2)) \), with \( n \) being the number of descriptors. If not specified otherwise, the times are also given in milliseconds per 512 descriptors.

All speed tests are done on a computer with a Intel i7 CPU @ 2.80GHz. We used the population count instruction in all cases. This instruction makes it possible to do fast distance calculations for binary vectors (see [6]).

4.1 Filtering

In this section we investigate the influence of the filters on the speed and performance. Section 4.1.1 investigate the optimal number of filters. 4.1.2 shows how dense and in what area these filters should be applied. Finally, section 4.1.3 shows, that gradient filters lead to the best performance.
4.1.1 Amount of Filters

In this section we investigate the influence of the number of filters on the speed and performance of our descriptor. With the generalization introduced in section 3.1.2 it would be possible to use any number of filters. Since codebook size increases exponentially, more than four filters are not usable in practice. Figure 4.1 shows how the dimensionality increases with the number of filters. The description time is linear $O(d)$ with $d$ being the number of dimensions. But for the Sparse Quantization a sorting of the descriptors value is necessary, which takes $O(d \times \log(d))$ time. As a result, using more than four filters is not practicable. For six filters, the quantization alone takes more than 100ms per 512 descriptors. Furthermore the matching of the descriptors takes more time. Calculating the distance between two descriptors is $O(d)$. But in matching all descriptors need to be compared against all the others, therefore matching takes $O(n^2)$ time, with $n$ being the number of descriptors to match. This fact makes it essential to have the fastest distance calculation possible.

In later experiments (such as in figure 4.5) we show that using four filters in BG-SIFT is enough to achieve the desired performance improvement. For four filters, the construction time of BG-SIFT is lower than SIFT (see table 4.1.1) and through the binarization it is four times faster in matching. Therefore we choose to use four filters. In section 4.3 we show, that the vector of G-SIFT with four filters can be quantized into a sparse binary vector without any loss of performance.

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Description time [ms]/512 descriptors</th>
<th>Matching time [ms]/512 descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>BG-SIFT 4-filters (KNN=1)</td>
<td>389.8</td>
<td>9.5</td>
</tr>
<tr>
<td>BG-SIFT 6-filters (KNN=1):</td>
<td>668.9</td>
<td>77.1</td>
</tr>
<tr>
<td>Vedaldi SIFT</td>
<td>534.2</td>
<td>39.6</td>
</tr>
</tbody>
</table>

Table 4.1: Speed of BG-SIFT compared to SIFT
CHAPTER 4. EXPERIMENTS AND RESULTS

4.1.2 Choice of the filter region

Here, we show experiments on the patch size, which is used to build the descriptor and the density, in which these filters are applied. Based on the shape of the filters we use (see section 3.1.1) one could assume, that applying these filters densely is not worth its computational cost. Dense, in this context means calculating the filter response for every pixel. Instead of doing this, we tested the performance, when computing the filter only for every other pixel (i.e. a step size of $\delta = 2$).

As a first step, we optimized the patch size for BG-SIFT and less-dense BG-SIFT ($\delta = 2$) over Yosemite (10,000 pairs) in figure 4.2. Then, we tested the performance with this optimized patch sizes on Liberty and Yosemite and compared the speed of them (figure 4.3). Furthermore we compared the performance of the less dense BG-SIFT with $\delta = 2$ on the test images of Mikolajczyk. Note that we used a fixed patch size of 48x48 pixels for that dataset. The results are shown in figure 4.5. Table 4.2 furthermore shows a comparison of the speed for the different steps of the descriptor construction, for the scale-aware version. As you can see, the less-dense BG-SIFT still performs better than SIFT, while being quite fast for such a high performance descriptor. This makes it suitable for real-time tasks such as Augmented Reality.

We tested the same filtering, with $\delta = 2$ on our implementation of SIFT. To preserve the dense representation of the patch, we changed the gradients of SIFT and used:

\[
g_x(x, y) = I(x + 2, y) - I(x - 2, y) \\
g_y(x, y) = I(x, y + 2) - I(x, y - 2)
\]

Figure 4.4 shows the results of this less dense SIFT. Furthermore table 4.2 shows the speed savings compared to SIFT.

4.1.3 Learning filters

Here we present the results of the two learning algorithms we used.

Figure 4.2: BG-SIFT dense and less dense. (a) Error rate at 90% (b) Speed (fixed-scale)
Learning based on performance

Here we present the results of the simple learning algorithm presented in section 3.3.1. As initial filters we used the filters shown in figure 4.6 (a). The best performing filters are shown in 4.6 (b). They improved the performance on the test set (the Mikolajczyk dataset) by 1%. Unfortunately the training set was too small, which lead to an overfitting. The learned filters do not perform well on other datasets such as the ones of Brown. But the learning showed that it is reasonable to choose larger gradients, i.e. instead of having a $\delta = 1$, which was our initial filter, we empirically found, that the use of a $\delta = 4$ is performing better. There still lies potential in improving the current filters, when learned on a larger and more diverse training set.

Figure 4.6 shows the initial, the learned and the final filter, which we use for all our performance tests. Through the use of this filter we could improve the performance of our descriptor by about 1.5% on the Liberty dataset with the BRIEF test setup (final results in section 4.8.1).

Learning based on the Sparsity criterion

We implemented an algorithm for Sparsity Learning. While our algorithm was able to optimize the sparsity, this did not lead to better results yet. This is probably due to the fact, that we were limited by the number of filters to use. Instead of learning a high number of filters (64 in the case of [22]) like Ng and others did, we learned only four or six filters. The reason for this is that we assign the filter responses to a codebook, as described before. Therefore four filters lead to a codebook of 80 elements, six filter to a codebook of 728 elements. Ng et al. follow a different path: They try to represent a patch directly by a linear combination of the learned filters.

It would be necessary to learn more filters and improve the codebook assignment step or replace it by a step that represents the patch by a linear combination of the learned filters. Figure 4.7 shows a performance comparison between BG-SIFT with the learned filters, BG-SIFT with default parameters and BRIEF. One advantage of the learned filters is their sparsity. We found 40-sparsity, i.e. 40 non-zero dimensions performing best. Therefore the resulting descriptor could be represented as a sparse vector in memory, storing only the positions of the non-zero dimensions, which needs half the memory (40 times 2 bytes),
CHAPTER 4. EXPERIMENTS AND RESULTS

Figure 4.4: SIFT dense and less dense. (a) Performance on Liberty (b) Performance on Yosemite

compared to the default, less sparse descriptor.

4.2 Codebook

SIFT uses a 2-dimensional uniformly distributed codebook of size 8. We experimented with various codebooks, but found that the influence of the codebook is minimal. Even a random codebook performs reasonable (see figure 4.9). But in order to prevent calculating the distance of a filter response to all the codebook elements, we decided to use a \( \bigcup_{k=1}^{4} B^4_k \) codebook, which allows for the speed optimization described in section 3.1.2. Figure 4.8 shows the speed improvement, when eliminating unnecessary distance calculations. Since the description vector is high dimensional, we tested the performance when taking a partial codebook. This is done by assigning a filter response to its nearest neighbor in \( \bigcup_{k<n} B^4_k \). I.e. we code a filter response vector \( (g_1, g_2, g_3, g_4)^t \) by its k strongest responses. We tested

\[
C_{80} = \bigcup_{k=1}^{4} B^4_k, C_{64} = \bigcup_{k=1}^{3} B^4_k, C_{32} = \bigcup_{k=1}^{2} B^4_k \text{ and } C_{8} = \bigcup_{k=1}^{1} B^4_k.
\]

Figure 4.10 shows the performance loss when using such a partial codebook. We tested the performance of G-SIFT, not BG-SIFT, as we would have to optimize the k for all the codebook sizes (see 4.3). Using only 64 or 32 elements could be interesting, considering the smaller use in memory and the faster matching times. For the reason of simplicity these versions of the descriptor are not tested any further in this thesis. For all experiments we use a full codebook with 80 elements.
Figure 4.5: Performance of BG-SIFT and less dense BG-SIFT (delta=2) on the Mikolajczyk dataset

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Filtering</th>
<th>Coding</th>
<th>Pooling</th>
<th>Sparse Quantization</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>BG-SIFT</td>
<td>8.3</td>
<td>149.7</td>
<td>203.1</td>
<td>27.2</td>
<td>388.3</td>
</tr>
<tr>
<td>BG-SIFT delta=2</td>
<td>1.8</td>
<td>62.6</td>
<td>56.5</td>
<td>19.3</td>
<td>140.3</td>
</tr>
<tr>
<td>SIFT like</td>
<td>4.3</td>
<td>166.3</td>
<td>339.8</td>
<td>0</td>
<td>510.4</td>
</tr>
<tr>
<td>SIFT delta=2</td>
<td>1.3</td>
<td>51.9</td>
<td>89.9</td>
<td>0</td>
<td>143.0</td>
</tr>
</tbody>
</table>

Table 4.2: Description speed of BG-SIFT (scale-aware) and less dense BG-SIFT (delta=2).
CHAPTER 4. EXPERIMENTS AND RESULTS

Figure 4.6: Filters for dense filtering. (a) Initial filters (b) learned filters (c) filters that generally offer a well performance. **Red:** Current pixel **Blue:** Positive part **Green:** Negative part

4.3 Binary Sparse Quantization

This step is essential for creating a descriptor that offers a good matching speed and a small memory need. We tested Sparse Binary Quantization on SIFT and on our new descriptor G-SIFT. In order to find the optimal $k$ (number of non zero dimensions) for the $k$-sparse binary vector, we optimized the 90% error rate. Figure 4.11 shows the optimal $k$ for the respective descriptors. For SIFT we found $k = 52$ performing best and for BG-SIFT $k = 247$.

We then tested the performance of binary SIFT and BG-SIFT on the datasets of Brown and of Mikolajczyk (figure 4.12 and 4.13), for $KNN = 2$ and $KNN = 1$. As you can see SIFT generally losses in performance when binarized with Sparse Quantization. But interestingly, the high dimensional G-SIFT can even benefit from such a quantization. Please note, that for matching, not the ROC 95% error rate is significant for the performance, but lower error rates, in the range of ROC 80% to ROC 90%.

4.4 Influence of KNN and Beta

In SIFT, a filter response vector is assigned to its two-nearest neighbors in the codebook (KNN=2). For our descriptor, we tested $k$-NN with $k \in \{1..4\}$. We found that a KNN with $k > 2$ does not improve the performance. Therefore we used KNN-2 and optimized over the distance factor $\beta$ (see section 3.1.2). Figure 4.14 shows the 90% error rate of BG-SIFT on the Yosemite dataset. We found that $\beta = 0.000255291$ is performing best. We compared this optimized KNN=2 version of the descriptor against one using KNN=1 in figure 4.15 and table 4.3. Note that we optimized over a training set of 10’000 pairs of the Yosemite dataset. Since there is no improvement on the Liberty test set, when using KNN=2 instead of KNN=1, we advocate the use of KNN=1. With the use of KNN=1, the calculation of an exponential weight is needless, which results in an enhanced speed of our descriptor. This speedup makes the BG-SIFT descriptor about 20% faster than with KNN=2. See table 4.4.
4.5 Pooling

In figure 4.16 we compare average against max pooling. As you can see, average pooling performs slightly better. The two strategies are equal in computation time. Therefore we use average pooling for the default configuration of BG-SIFT.

4.6 Smoothing

Here, we compare the performance of the descriptor, without any smoothing, with a simple blur (as in BRIEF) using a kernel size of 5 and with a Gaussian smoothing with an optimized sigma. We optimized over Yosemite with 10’000 pairs and found $\sigma = 1.3$ performing best (figure 4.17).

While our descriptor performs well without any pre-smoothing, other descriptors are very dependent on it. In the paper of BRIEF [6], Calonder et al. compare BRIEF to SIFT on the test sets of Mikolajczyk. There they show recognition rates for BRIEF that are higher than SIFT, mainly on images that contain blur (such
as the Trees image set). We could not reconfirm this results. SIFT uses a base smoothing of $\sigma_0$. This smoothing is the minimum smoothing applied on the input image. In the case of upright and unscaled SIFT, we use this base smoothing $\sigma_0 = 3.2$. This follows the implementation of Vedaldi, where $\sigma_0 = 1.6 \times 2^{(1/S)}$ with $S$ being the number of scales. For the use of unscaled keypoints we therefore use $S = 1$. Other than what is shown in [6], SIFT performs better in any cases, using this parametrization. See figure 4.24

### 4.7 Variations of BRIEF

Here we present the results of the formulation described in Chapter 3 for the use of a descriptor that is similar to BRIEF, i.e. with no pooling, but concatenation. Experiments with various filters have shown that using more local tests applied at multiple locations lead to a better performance. We therefore used random filters with a Gaussian distribution, as in BRIEF but with four centers (see section 3.4).

Figure 4.19 shows the performance of G-BRIEF and figure 4.20 the speed and the used filters. Note that using pairs results in a higher performance, but the descriptor is four times longer, which results in a higher matching time.
4. EXPERIMENTS AND RESULTS

4.8 Comparison to the state-of-the-art

In this section we compare our descriptors to the state-of-the-art. Through the course of our experiments we made the following observations:

**Filters.** By increasing the number of filters used, the performance can be enhanced. Since this also increases the computation time, more than four filters are not useful. Four filters offer a performance increase, while the descriptor can still be computed faster than SIFT. Furthermore we observed, that dense filtering is only desirable, if the goal is maximum performance. If speed is an issue, it is not worth the computational cost (see table 4.2).

**Learning.** Through the learning of filters we could observe, that complex filters are not performing well. They are also long to compute. Therefore we used gradient filters for our final descriptors.

**Codebook.** The influence of the codebook choice is minor. It is best to choose a codebook that allows for codebook assignment via sorting. This optimization improves the speed of the codebook assignment step considerably. In the case of four filters, this step is done 7 times faster when using sorting. See section 3.1.2 for the sorting algorithm and figure 4.8 for the speed improvement.

**Binary Sparse Quantization.** The performance of the descriptor is almost constant for a large range of different $k$ (non-zero elements). This is important, in order to have an optimal configuration for any type of input image. Furthermore we could show that the Sparse Quantization works better with the use of a bigger codebook. While Binary Sparse Quantization is working well in the case of four filters, it does not perform for SIFT, which has only two filters and a codebook of eight elements. (see 4.13).

**KNN and Beta.** The influence of the choice of KNN is small. The use of KNN=2 with an optimized Beta can lead to a slightly improved performance. Unfortunately the optimal Beta is dependent on the input data and using KNN> 1 adds computational cost. Therefore we see it best to use KNN=1.

**Smoothing.** Thanks to the choice of our filter arrangement, BG-SIFT does not depend on pre-smoothing. It also performs very well on image patches without any pre-smoothing. With the use of an optimal smoothing BG-SIFT can gain some performance, though.

Smoothing is very important to other descriptors, such as BRIEF or SIFT. The authors of BRIEF present...
Figure 4.12: Performance of (Binary)-SIFT and (Binary)G-SIFT on the Yosemite dataset

Figure 4.13: Performance with and without binarization on images from the Mikolajczyk dataset

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Description Time [ms]/512 descriptors</th>
<th>Matching time [ms]/512 descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>BG-SIFT 4-filters KNN=1</td>
<td>389.8</td>
<td>9.5</td>
</tr>
<tr>
<td>BG-SIFT 4-filters KNN=2</td>
<td>484.9</td>
<td>9.5</td>
</tr>
</tbody>
</table>

Table 4.4: Description and matching speed (scale-aware) for different KNN
Figure 4.14: Beta optimization of BG-SIFT with the 90% error rate

Figure 4.15: Performance of BG-SIFT with KNN=1 and KNN=2
results in their paper, that show BRIEF performing better than SIFT on blurred images[6]. We could not
reconfirm these results and see them as a suboptimal smoothing in their use of SIFT (see section 4.6).

As a result of these observation we see the following configurations the most interesting, which we compare
in the remainder of this section:

**BG-SIFT.** This is the BG-SIFT with 4 filters in the default configuration. Default means: \( KNN=1 \), average
pooling, patch size = 58. dense filtering. This configuration is high performing, even thought, in some cases,
BG-SIFT-4 with \( KNN=2 \) is performing slightly better. We advocate to use the configuration with \( KNN=1 \)
as it is about 20% lower in description time.

**BG-SIFT less dense.** This is the BG-SIFT with 4 filters but the filters are applied with a step size \( \text{delta} = 2 \).
Furthermore we use patch size = 56. We use average pooling and \( KNN=1 \) as above. We find this
configuration particularly interesting, because it still outperforms SIFT, while being four times faster than it
in both, description and matching.

**G-SIFT siftlike.** This is our descriptor in the configuration of SIFT. That is: two filters, codebook size
\( C = 8 \), \( KNN=2 \) and no binarization. This descriptor we present in order to compare our implementation
of SIFT to the standard implementation. Other than SIFT, we do not use an arc tangent, but sorting and
distance computations (see section 3.1.2).

Section 4.8.1 uses the datasets and experimental setup of Brown[5] and further reproduces the Liberty test
used in BRIEF[6]. Section 4.8.2 tests the Matching performance of our descriptors on the Mikolajczyk
datasets and compares it to SIFT and BRIEF. Finally we show speed tests in section 4.8.3.

Last but not least, section 4.8.4 compares our descriptor parametrized like SIFT to the Vedaldi implementa-
tion of SIFT.
CHAPTER 4. EXPERIMENTS AND RESULTS

Figure 4.17: Influence of pre-smoothing on yosemite

Figure 4.18: Performance of BG-SIFT with different smoothing strategies
4.8.1 Multi-view Stereo Correspondence Dataset

Brown test setup

As described at the beginning of this Chapter, these datasets contain 400k patches, from which use a test set of 10’000 or 100’000 pairs with 50% matching patches and 50% non-matching patches in all cases. The dataset and the size of the test set are specified in each case in the title of the figure. We compare the ROC-curve of BG-SIFT to SIFT in figure 4.21. Furthermore we compare the 95% error rate to the ones obtained by Simonyan et al.[19] and Brown[5] in table 4.5. For the results of Brown we include the results of their parametric descriptors. Note that we do not compare to the results of Brown, achieved through parametric optimization followed by a Principal Component Analysis (PCA). PCA improves the results, but it also adds computational cost and PCA was not applied here either.

Our descriptors outperform SIFT on all datasets. In the best configuration our descriptor is also a bit better than the one of Brown. This is even despite the fact that no optimization of the spatial pooling was used. Furthermore the descriptor is binary, which makes it orders of magnitude faster in matching (Brown’s descriptors have 400 or more real-value dimensions). Simonyan et al. achieve better results on all datasets. As in the paper of Brown, they get their improvement over SIFT from optimizing the spatial pooling. Note, however, that they present their results only in terms of the false positive rate at 95%, which is not a reliable criterion to make a good performing descriptor for matching, as our experiments have shown. Furthermore Browns and Simonyans descriptors are very complex and high-dimensional (or their dimensionality was reduced by a costly PCA). Therefore they are not suitable for time critical tasks.

To achieve these results we used the following parameters: Gaussian pre-smoothing with $\sigma = 1.3$, patch size 58 (for less dense: patch size 56) and $k = 247$ for the $k$-sparse binary vector (number of non-zero elements in the descriptor).
Figure 4.20: Speed and filters of G-BRIEF. (a) Speed per 512 descriptors (b) Four centered filters

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yosemite</td>
<td>15.70%</td>
<td>16.76%</td>
<td>28.50%</td>
<td>15.91%</td>
<td>10.65%</td>
<td>53.63%</td>
</tr>
<tr>
<td>Liberty:</td>
<td>18.20%</td>
<td>19.83%</td>
<td>35.09%</td>
<td>20.48%</td>
<td>18.47% / 17.81%</td>
<td>57.15%</td>
</tr>
<tr>
<td>Notre Dame:</td>
<td>14.44%</td>
<td>15.64%</td>
<td>26.10%</td>
<td>14.43%</td>
<td>9.71%</td>
<td>50.96%</td>
</tr>
</tbody>
</table>

Table 4.5: False positive rate at 95% recall on Browns datasets

BRIEF test setup

This experiment is comparable to the one from BRIEF[6]. Note, that the procedure is slightly different, which results in a performance that is generally a few percent lower for all the descriptors in our experimental setting.

Generally, binary descriptors loose in performance, if used in large-scale matching. In order to test the behavior of BG-SIFT we therefore further tested it on the Liberty dataset\(^1\) of Brown. From this dataset we extract a subset and match one-against-all. Every patch has a real-world point associated. When choosing the patches, only two patches with the same real-world point are chosen, so that there exists only one correct match. For our tests we used 20'000 of those patches. These we split in \(k\) subsets with \(k = 20000/n\) where \(n\) is the number of patches matches against each other. \(n\) is ranging from 500 to 2000 patches, as shown in figure 4.22. A patch is considered to be matched correctly, if its corresponding patch (same real-world point) is its nearest-neighbor.

As you can see, BG-SIFT performs about 2% better than SIFT. Furthermore it outperforms BRIEF-32 (the default configuration) by about 14%. Even when using BRIEF-128 (1024 dimensions) the performance gap to BG-SIFT (1280 dimensions) is still about 10% (see table 4.6).

\(^1\) http://www.cs.ubc.ca/~brown/patchdata/patchdata.html
CHAPTER 4. EXPERIMENTS AND RESULTS

Figure 4.21: ROC-curves for the performance with the Brown test setup

**Left:** Yosemite **Right:** Liberty

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>40*500 patches</th>
<th>20*1000 patches</th>
<th>10*2000 patches</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIFT</td>
<td>56.01%</td>
<td>48.94%</td>
<td>42.79%</td>
</tr>
<tr>
<td>BG-SIFT</td>
<td>58.80%</td>
<td>51.47%</td>
<td>45.40%</td>
</tr>
<tr>
<td>BG-SIFT delta=2</td>
<td>52.19%</td>
<td>44.93%</td>
<td>39.07%</td>
</tr>
<tr>
<td>BRIEF-32</td>
<td>40.61%</td>
<td>34.09%</td>
<td>28.49%</td>
</tr>
<tr>
<td>BRIEF-128</td>
<td>44.48%</td>
<td>37.81%</td>
<td>32.13%</td>
</tr>
</tbody>
</table>

Table 4.6: Performance on the Liberty dataset

4.8.2 Mikolajczyk dataset

For this test we use keypoints and images created with the Vedaldi implementation of SIFT. SIFT does a pre-smoothing on the image and then creates images with different smoothings and scalings for the different scales of the keypoints. A descriptor is calculated on the image, that corresponds to its scale. Through the use of this data, we avoid the recalculation of the scaled images. But this also means that this test is optimized for the SIFT descriptor and not for our own.

First, in figure 4.24 we compare the upright and unscaled version of BG-SIFT to SIFT and BRIEF and the scaled and rotated versions of BG-SIFT and SIFT. This is done on image sets with viewpoint changes, blur, JPEG artifacts and illumination changes. Furthermore we test the invariance of SIFT and BG-SIFT to scale and rotation of two image sets in figure 4.23. BG-SIFT looses its advantage over SIFT when tested for scale and rotation invariance. This is due to the reasons described before and to the fact, that the rotation cannot be encoded as nicely as in SIFT. Since SIFT works with the angle to do the codebook assignment, it simply uses the angle relative to the main orientation, as provided by the detector. In the case of BG-SIFT, this is done differently. When using the rotation-aware version, the filters are turned by the angle of orientation of the keypoint. This introduces a numerical error, as the filters are applied at pixel level. As a
result, rotation works slightly worse than for SIFT.

4.8.3 Speed comparison

In this section we compare the speed of the different descriptors in construction and matching. Table 4.7 and figure 4.25 show the time for constructing and matching 512 descriptors. BG-SIFT in the default configuration is slightly faster than SIFT in description and four times faster in matching.

Less dense BG-SIFT is much faster than SIFT in both, description and matching, while retaining a better performance (see figure 4.21). In figure 4.26 we test the potential of BG-SIFT in large-scale matching. Since description time is done in linear time ($O(n)$) and matching time is quadratic ($O(n^2)$), BG-SIFT is especially useful for large scale matching problems.

4.8.4 SIFT parametrization

Here we compare our implementation of SIFT to the Vedaldi implementation. This is done in figure 4.27. So far we were not able to achieve exactly the same performance as in SIFT. This is probably due to a different pre-smoothing and weighting in the codebook assignment. The optimization of these parameters was not further examined, yet. But while the performance is slightly lower than in the Vedaldi implementation, our implementation is slightly faster. (see Table 4.7) The reason for this is, that SIFT needs to calculate the arc tangent, which is very expensive.
4.9 3D Reconstruction

In order to test the descriptor in a real-world setting, I developed a 3D reconstruction program with Structure from Motion. It is based on the paper “Based on What and where: 3D object recognition with accurate pose” by Gorden and Lowe[7] and the SBA package[12] for Bundle Adjustment. The pipeline has the following steps:

1. **Feature extraction.** Detect and describe keypoints from the input images.

2. **Multiview correspondences.** Build matches across multiple images, i.e. find 2D points, which belong to the same 3D point.

3. **Initialization.** Make an initial assumption for the 3D points position.

4. **Bundle Adjustment.** Refinement is used to improve the assumed 3D point positions. Bundle Adjustment does a least-square minimization of the reprojection error of those points.

4.9.1 Feature extraction

This step is straight forward. SIFT keypoints are detected using the Vedaldi implementation. These are then described using the default implementation of BG-SIFT.

4.9.2 Multiview correspondences

As in [7], we build multiview correspondences from pairs. Let \( f_{hi} \) be the \( i \)-th feature of image \( h \), with \( i \in \mathcal{K}_h \). All the images of the dataset are matched in pairs of two neighboring images, i.e. we match \( f_{hi} \) to \( f_{(h+1)j} \) \( \forall i, j \) with \( i \in \mathcal{K}_h, j \in \mathcal{K}_{h+1} \). In order to have a highly reliable matching, this is done using Ratio matching (see section 3.2). After these correspondences have been established, they can be connected to...
multiview correspondences by connecting the keypoints that share a feature. So, if feature \( f_{1i} \) is matched to feature \( f_{2j} \) and \( f_{2j} \) to \( f_{3k} \) (and so on) all these keypoints are assigned to the same 3D point.

### 4.9.3 Initialization

While Gorden and Lowe claim that Bundle Adjustment can be simply initialized by placing all the cameras at the same point and all the 3D points on a plane directly faced by those cameras (see [7]), this did not work in our case. The problem with Bundle Adjustment is that it risks falling into a local minimum, if not provided with a good initialization. But “This area is still to some extend a black art”, as Andrew Zisserman states in this book “Multiple View Geometry in Computer Vision” [9].

How the algorithm should be initialized is dependent on the true camera positions. Other than Gorden and Lowe, we use the CMU Grocery Dataset used in [10] and available online\(^1\). This dataset offers images of Grocery objects, taken from an all around, 360\(^\circ\) view. Therefore we found a different initialization, which showed to work in this setting: All cameras were placed on a circle around \((0, 0, 0)\), facing \((0, 0, 0)\). The 3D points \(X_i\) are all placed at \((x_{ij}/1000000, y_{ij}/1000000, 0)\), where \(x_{ij}\) and \(y_{ij}\) are the projection of \(X_i\) in image \(j\). For \(j\) we use the first image a 3D point appears.

\(^1\)http://www.cs.cmu.edu/~ehsiao/datasets.html
CHAPTER 4. EXPERIMENTS AND RESULTS

**Table 4.7: Speed for description (scale-aware) and matching**

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Description Time [ms/512 descriptors]</th>
<th>Matching time [ms/512 descriptors]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BG-SIFT 4-filters KNN=1:</td>
<td>389.8</td>
<td>9.5</td>
</tr>
<tr>
<td>BG-SIFT 4-filters KNN=1 (fixed-scale):</td>
<td>211.4</td>
<td>9.5</td>
</tr>
<tr>
<td>BG-SIFT 4-filters KNN=1, delta=2:</td>
<td>140.3</td>
<td>9.5</td>
</tr>
<tr>
<td>SIFT-like</td>
<td>510.4</td>
<td>39.6</td>
</tr>
<tr>
<td>Vedaldi SIFT</td>
<td>534.2</td>
<td>39.6</td>
</tr>
<tr>
<td>BRIEF-256</td>
<td>2.0</td>
<td>1.8</td>
</tr>
</tbody>
</table>

Figure 4.25: Construction speed of the different descriptor parameters (scale-aware)
4.9.4 Bundle Adjustment

In order to refine the initial estimate, which is of course a very poor approximation of the real 3D structure we apply Bundle Adjustment (BA). BA is a least-square minimization of the reprojection error of the assumed 3D point positions. In our case we work with calibrated cameras, i.e. with known camera intrinsics. Since projections of a 3D point in multiple images are known, it is possible to estimate its 3D position and the position and movement of the camera. For this minimization the ”Sparse Bundle Adjustment“ (SBA) of Lourakis and Argyros[12] was used. SBA is a C++ framework for large scale bundle adjustment based on a Levenberg-Marquard minimization. Figure 4.28 shows the result of the 3D reconstruction of a juicebox.

As you can see, there are some outliers, which are most likely incorrect matches from the multiview correspondences phase. In order to improve the 3D reconstruction, this outliers should be removed and the bundle adjustment reiterated. This is not done here, but would be of great value for a more accurate reconstruction.
CHAPTER 4. EXPERIMENTS AND RESULTS

Figure 4.26: Speed for description (scale-aware) and matching

Figure 4.27: Performance comparison of Vedaldi SIFT to our implementation of SIFT
Figure 4.28: 3D reconstructions of household objects

(a) An image of the juicebox set (b) & (c) Above and side view of its 3D reconstruction.  
(d) An image of the can set (e) & (f) Above and side view of its 3D reconstruction.
Chapter 5

Conclusion

This thesis presented a new binary descriptor, named BG-SIFT. This descriptor is based on a novel formulation introduced by Boix et al.[4]. While most common descriptors are based on a histogram of oriented gradients, this formulation allows for a more general use of gradient filters and a more sophisticated encoding. Through the exploitation of this encoding it was possible to develop this new descriptor, which outperforms SIFT and other state-of-the-art descriptors.

With our experiments we have shown, that increasing the number of gradient filters and using a more elaborate codebook leads to an improved performance. Furthermore we observed that applying the filters densely is not worth the computational cost, except if highest performance is desired. While we could not achieve final results yet, first experiments with filter learning have shown, that it is possible to find more suitable filters, which can lead to a performance improvement.

With our experiments we showed that Binary Sparse Quantization introduces less error when used with larger codebooks. Other parameters, such as the number of nearest neighbors or the pre-smoothing have only a minor influence on the performance and speed of the descriptor. For a complete list of findings see section 4.8.

As a result of our findings we propose to use two configurations of our descriptor:

**BG-SIFT.** This is our default descriptor. For this configuration we use a KNN=1, which we view as the best trade-off between performance and speed. This descriptor is outperforming SIFT and achieves state-of-the-art performance on the widely used datasets of Brown. In this configuration our descriptor is about 25% faster than SIFT in the description phase and four times faster in matching. It is orders of magnitude faster than any other method with this performance. Other state-of-the-art descriptors use a complex spatial pooling, which is long to compute and results in a high-dimensional floating-point vector, which is computationally expensive to compare.

**BG-SIFT less dense.** This is a fast variant of our descriptor, which is usable for real-time applications. It needs less than half the time for description, compared to the default configuration, but is only 1-2% worse in terms of false positive rate at 95% recall (SIFT has 12-17% higher error rates). The scale-aware version is build and matched in 150 milliseconds per 512 descriptors. The non-scaled version even needs as little as 90 milliseconds.

The focus of this work lay on developing a binary descriptor that performs better than SIFT, but it is also possible to use the introduced formulation to develop very fast descriptors, similar to BRIEF. Currently we
are working on such a descriptor and first experiments, presented in section 4.7, show that it is possible to develop a descriptor that is built as fast as BRIEF, while offering an increased performance.

While we gain our improvement from a more sophisticated encoding, other researchers worked on improving the spatial pooling. Brown[5] proposed a framework, which learns descriptors that outperform SIFT and perform almost as good as our descriptor. Furthermore Simonyan et al.[19] presented a learning framework, which uses Convex Optimization in order to improve the results of Brown. With this framework they learned descriptors, that outperform the results of Brown and also our own.

Both methods use the filtering and codebook assignment of SIFT, but are using more and differently distributed spatial bins. There lies much potential in applying a similar pooling on top of our descriptor to further improve its performance. This is the direction my future work will be go to. Furthermore the learning framework will be improved to use larger and more diverse training sets. In these measures we see a lot of potential for further improving our descriptor.
Bibliography


