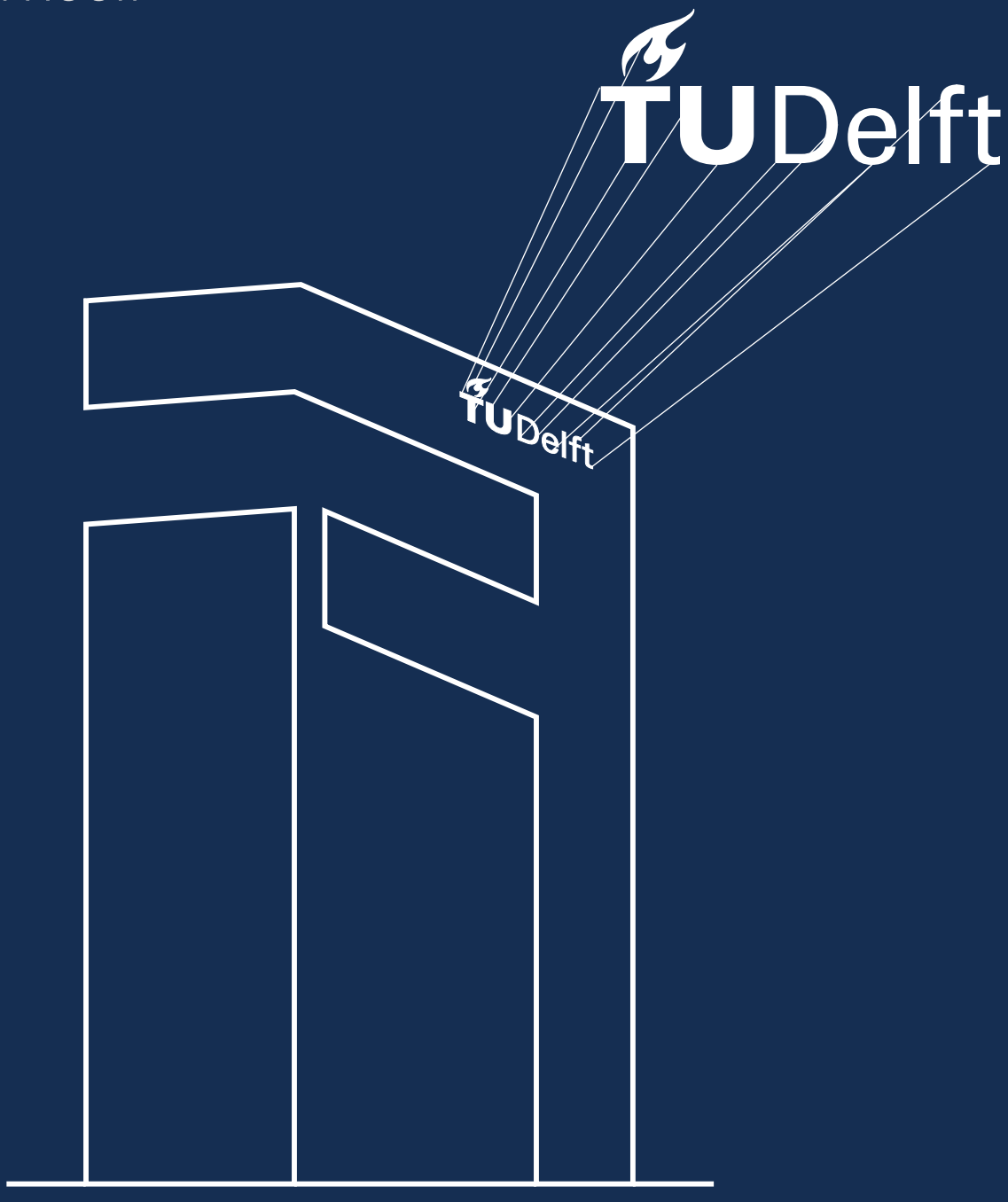
Image Search Engine for Digital History

A Standard approach

Max Deutman, Philip Groet, and
Owen van Hooff



Engineering Historical Memory – Bachelor Thesis 2021

TU Delft

Image Search Engine for Digital History

Standard Approach

by

M. Deutman, O. van Hooff, P.M.Q. Groet

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended on Wednesday June 30, 2021 at 09:00 AM.

Student numbers: 4592743, 4786173, 4569296
Project duration: April 19, 2021 – July 2, 2021
Thesis committee:

Prof. dr. ir. A.H.M. Smets	TU Delft, Chair of the Jury
Dr. ir. J. Dauwels	TU Delft, Supervisor
Dr. S. Cotofana	TU Delft
MSc. B. Abdikivanani	TU Delft

TU Delft

Abstract

This report details the evaluation of current image matching implementations for the use in an image search engine, specifically for digital history. Due to the vastness of historical (digital) libraries this search engine must be able to search all (inter)national databases with equal performance. Current search engines use linguistic keywords to describe an image and search for others, introducing a language bias. This project focuses on image-to-image matching, bypassing language altogether.

This report only addresses image matching algorithms based purely on mathematics, no machine learning is addressed within this thesis. This report will cover the performance and usefulness of template matching, ORB feature extraction, SIFT feature extraction, SURF feature extraction, Brute-Force matching, and FLANN matching. Machine learning algorithms for the use in an image search engine are addressed by the other subgroup of this project [1].

Preface

This thesis is written as part of the Bachelor End Project [EE3L11] 2020-2021. The project ”Image Search Engine for Digital History” was proposed by Justin Dauwels in collaboration with Andrea Nanetti and *Engineering Historical Memory* (EHM) [2]. EHM is an ongoing research project which has the goal of organizing and delivering historical knowledge in the digital age. The research presented in this thesis will contribute to this project by starting the development of an unbiased image search engine to navigate this historical data. Within the 10 weeks of this project we have shown the possibilities of image-to-image matching and have developed a prototype of a working image search engine.

We would like to thank Justin Dauwels, Andrea Nanetti, and Ioan Lager for their time and guidance during this project. We are thrilled that our research will benefit the *Engineering Historical Memory* project and that we were allowed to be a part of this collaboration [3].

M. Deutman, O. van Hooff, P.M.Q. Groet
Delft, July 15, 2021

Contents

1 Introduction

This thesis, in a wider perspective, is part of an ongoing research project *Engineering Historical Memory* [2] aiming to develop and asses applications for the (re)organisation and delivery of global historical knowledge in the digital age. Nanetti explains in his paper [4] the need for a multilingual and trans-cultural approach to encoding/decoding human experience and transmitting this to the next generations of humanity. Advances in information technologies such as artificial intelligence and machine learning algorithms can assist in (i) decoding knowledge and wisdom embedded in cultural artefacts and social rituals, (ii) encoding data in machine-readable systems, (iii) aggregating information according to the user's needs in real time, and (iv) simulating the effects of erasing, neglecting, preserving, and sharing human experiences.

To contribute to this need, an image search engine for digital history is developed. This thesis describes current state-of-the-art information technologies and includes the search engine design process. Section 1.1 covers the analysis and framing of the problem. Additionally, it includes the analysis and discussions necessary for scoping, bounding, and creating the Program of Requirements described in Chapter 2. Chapter 3 focuses on the existing technologies that have the potential of forming the building blocks of the search engine. In Chapter 4 the design process is discussed, specifying all decisions made on basis of the Program of Requirements. Chapters 5 and 6 will include the prototype results, project wide conclusions, discussions, and recommendations that emerged from this research. Additional analysis and explanations relevant in the process of this project, are described in Appendix A and C.

1.1. Problem Definition

Defining the problem is crucial and requires the assessment of the context and scope. The predetermined project conditions are the ten weeks of available time, the group size consisting of six electrical engineering students, and the supervision by J. Dauwels and A. Nanetti. Additionally, some existing suggestions were provided at the start by the supervisors: a proposal document, a short description on template matching, and links to potentially useful Python libraries.

To obtain a better understanding of the underlying factors and find potential levers for decision making, an inductive logic tree has been created. Based on this, the decision was made to investigate existing historical applications, object detection concepts, image extraction and matching methodologies, libraries, tools, and representative datasets.

1.1.1. Problem Analysis

Looking into existing historical applications, relatively little was found on applying image search and retrieval in a historical context. This indicates research potential for developing an image search engine for digital history. The results found include the search of reproductions of art through visual attributes for historians [5] [6], detecting lost heritage in historical video material [7], word-image classification in historical document collections [8] [9] [10], indexing expert image collections specifically on heritage

image datasets [11], the MARS (Multimedia Analysis and Retrieval System) used on images of ancient African artifacts from the Fowler Museum of Cultural History at UCLA [12], and lastly the search for artistic connections across cultures using image retrieval [13]. The specific application of using image retrieval for improving historical research is most similar to [5] and [13]. Particularly [13] is interesting, as it (i) finds pairs of semantically related artworks that span different cultures, media, and millennia, (ii) builds on and improves current approaches in image retrieval, and (iii) has been implemented online. However the algorithm distinguishes object media: objects may differ in material. Additionally, it uses filters based on human interpretation to structure content, which is not of interest.

Zooming in on object detection, extraction, and matching; several important ideas, methodologies, and concepts have been found. One of these is Content-Based Image Retrieval (CBIR): using color, shape, and texture of one image to find similarities in another images [14]. Experiments were performed on huge image databases and major performance increases were obtained after involving neural networks [15]. Besides color, shape, and texture; according to [16], images with variations in viewing angle should also be taken into account. However, the solution proposed in this paper does not provide leverage because it requires labelling of data and object specification. In [17] a method is explained on how to retrieve objects from a large corpus, and accomplishes this through improving the visual vocabulary and incorporating spatial information into the ranking. This is an interesting approach to improve speed in large amounts of data. However, using a visual vocabulary, a collection of visual words which together can give information about the meaning of the image (or parts of it) [18], is out of scope for this project. The 'image meaning' should not be involved in the engine, because it inherently puts human interpretation into the machine. This creates, although arguable, problems that this thesis considers out of scope. Some of the problems are elaborated in the Program of Requirements 2 and the paper written about the Ethics and Technology of search engines [19]. Methodologies to detect object types within an image include both *Standard* (handcrafted) approaches and *Deep Learning* approaches [15] [20] [21] [22].

Regarding image matching, two categories exist within the 'standard' realm: template-based matching and feature-based matching. Template matching is a machine vision technique that identifies parts of an image that match a predefined template pixel by pixel. Advanced template matching algorithms find occurrences of the template regardless of their orientation and local brightness [23] [24]. Feature based matching is used when both source and template images contain more correspondence with respect to features and control points [25]. Image features such as edges and interest points provide rich information on the image content. These features are unique for each image and hence, help in identifying the similarities between images. The features of an image are not affected by change in size and orientation, and are therefore suitable to 'identify' images that have been transformed in some fashion. Additionally, this approach is more efficient if the image has a large resolution: Moving a template image across a large source image, one pixel at a time, and repeating this process at different scales is computationally expensive [25].

1.1.2. Problem Scoping and Bounding

The necessity of this project lies in the search for visual content in the increasing amount of digital content. Nanetti explains in his paper [4] the need for a multilingual and trans-cultural approach to decoding-encoding human experience and transmitting this to the next generations of humanity The additionally of this project is the ability for historians and heritage stakeholders to find information and starting points for research through exploration of visual content. Special care is taken to avoiding text and 'image meaning' as a way of describing an image. Additionally, the risks of not conducting such research are, hypothetically speaking, loss of information and access to it across the globe.

The first objective is developing an image search engine for digital history, capable of retrieving images from various databases (e.g. Wikipedia, Europeana) similar to the user input image. A second objective is to write theses that elaborate on the approaches and results of creating such an image search engine. Additionally, business and ethics considerations will be included in separate documents and presentations. The first objective is completed if the engine (i) extracts and matches images, (ii) can retrieve images from image databases, and (iii) is not based on meta-data. The second objective is completed according to the manual [26].

The constraints of this project are the 10 weeks of available time for both objectives, the use of only

python related libraries or tools. Moreover, no object recognition and error feedback is implemented.

The parameters of the image search engine are the underlying thresholds used by the algorithm. These can be used to improve or change the decision making of the algorithm on image similarity. Additional parameters are keypoints and matches detected by the algorithm. These (can) differ per methodology used. The parameters affect the key performance indicators: time, precision, recall, and (balanced) accuracy.

1.1.3. Problem Statement

Based on the analysis, scoping, and bounding, the project has been split into investigating *Standard* and *Deep Learning* methodologies. The following problem statement has been formulated for this thesis:

To develop an algorithm in ten weeks that uses template or feature-based image retrieval to match images in multiple formats.

2

Program of Requirements

The Program of Requirements elaborately defines the functionality of the image search engine. It consists of key performance indicators (KPIs) and the conditions that apply to the development and implementation of the image search engine for digital history. A distinction is made between mandatory requirements, trade-off requirements, functional requirements, and non-functional requirements. The full Program of Requirements is depicted in figure 2.1.

Mandatory requirements
Functional requirements
1. The system must find and return images that match to a user input image 2. Image matching is based solely on image content
Non-functional requirements
3. The software must be written in the same language as the existing codebase of Engineering Historical Memory 4. The software must be able to be inserted in the existing codebase of Engineering Historical Memory 5. Out of all returned images, at least 80% must be true positives (precision > 0.8) 6. Out of all images that are supposed to be matches, at least 25% must be found (recall > 0.25) 7. Balanced accuracy must be at least 70% 8. The software implementation must make use of libraries and functions free for academic use 9. The system must accept the common image codecs jpg and png 10. The system must support image files up to 10MB in size 11. The software must allow for parallel computing 12. The full implementation must be completed within 10 weeks by a group of 6 students 13. The software must be able to be tested on hardware accessible to the group

Trade-off requirements
Functional requirements
14. The search time should be as low as possible
Non-functional requirements
15. Precision should be as high as possible 16. Recall should be as high as possible 17. Balanced accuracy should be as high as possible 18. The codebase should be structured clearly and documented in such a way that others can continue on our work 19. The search engine should not be biased (should not include user feedback to improve the performance) 20. The system should show how and why matches were found 21. The supported number of image formats should be as high as possible

Figure 2.1: General Program of Requirements

Its core functional requirements, shown as items 1 and 2, follow directly from the proposal document and supervisor discussions. The qualities and attributes follow from discussion and existing literature.

Items 3, 4, and 8 account for further research by other scholars. Items 5, 6, and 7 are important for performance measuring purposes. Items 8, 9, 10, and 11 limit the possible implementation, and items 12 and 13 specify the product-project relation. The trade-off requirements specify what is desired. Item 14 follows from item 1 and 2 in a end-user perspective. Keeping the search time low will be kept in mind during the project, but as this project focuses on the search algorithm ensuring its best performance in terms of matching, time will be a more relevant constraint when EHM further develops the actual search engine. Search time is also very dependent on the available compute power, when writing these requirements it was reckoned that we would not have access to super computers and would thus not 'worry' too much if the compute time seemed unsatisfactory for the implementation in a functional image search engine. Items 15, 16, and 17 specify desired attributes for performance. Item 18 supports, again, further research and development. Items 19 and 20 specify the desire for transparency and understanding of underlying search engine decisions. This is especially important when considering ethical concerns, such as the topics discussed in the ethics papers written by our 2 subgroups of this project [27] [19]. Item 21 supports items 1, 2, and 9. Requirements specifically for a *Standard* approach are shown in figure 2.2. These items have been decided upon to limit the scope of the project and are based on literature study and supervisor discussions.

Standard
Functional requirements
22. The algorithm must be based purely on mathematical formulae and/or models
Non-functional requirements
23. The algorithm musthave threshold values that can be adjusted to alter the performance

Figure 2.2: *Standard* Program of Requirements

The requirements specified are focused on the performance, creation, efficiency, and product handling. Considerations about safety, environment, and cost are not included in the scope of this thesis due to available time and resources. This directly implies that such considerations are open to research. Ethical considerations are discussed in separate documents [27] [19] and includes concerns about artificial intelligence and search engines.

3

Analysis of Existing Methods

In this chapter some of the existing methods to match images will be analyzed. The existing methods can be divided in 2 categories. The first one being template matching, these methods look at the pixel correspondence between a template and a source image. Secondly, the other methods will fall into the category called feature matching. Methods in this category utilize more advanced ways of matching. Each category consists of a huge amount of different implementations. This chapter will discuss the most common implementations from the OpenCV Python library [28]. Section 3.1 will discuss some of the template matching variants, the feature matching methods are discussed in section 3.2.

3.1. Template Matching

Template matching [29] is the most simple form of image matching where a template image [wxh] is essentially dragged across a larger image [WxH] and for every position the pixel correspondence is analysed. It is a very basic pixel-to-pixel matcher for which multiple techniques can be used to decide how much the template pixels differ from the search image pixels in a given location. This thesis analyzed the methods available in the OpenCV library, these methods are:

- A.1 SQDIFF [Square Difference]
- A.2 SQDIFF_NORMED [Square Difference Normalized]
- A.3 CCORR [Cross Correlation]
- A.4 CCORR_NORMED [Cross Correlation Normalized]
- A.5 CCOEFF [Correlation Coefficient]
- A.6 CCOEFF_NORMED [Correlation Coefficient Normalized]

The mathematical formulae of these convolutions can be found in appendix A.1.1. In the formulae I denotes image, T template, and R result. The x and y represent the pixel coordinates (coordinates [0,0] are the top left corner). The best matches are based on global minimums (with SQDIFF) or maximums (with CORR or CCOEFF). To show this in practise, in figure 3.1 the left image has a very bright spot which corresponds to the best match location of the template. An in-depth example of template matching, including results, can be found in A.1.2.

It is inherent to the way template matching works that the template should be the exact size of the occurrence in the image to be searched. This is problematic as it is not known how big the object in the search database is beforehand. Therefore to match the template with an arbitrarily sized search image, the template image will be resized up and down in many steps, and the template matching process will be repeated for every size step. Downscaling and upscaling the image is not a very intensive operation, but having to redo the template matching process for every size is a very computationally intensive operation [25]. Also, preliminary tests of this technique have shown that even with a large

Figure 3.1: Example template matching. Source: https://docs.opencv.org/

amount of different size scales tried, the template is still not always found for 'easy' images. Code of this preliminary test can be found in appendix G.

For this project the template matching variants are not suited. This project has the goal of finding matching images that are not known beforehand. Template matching cannot achieve this as the template image needs to be from the image that it is searching in.

3.2. Feature Matching

Template matching performs very well when templates have no strong features within an image, since they operate directly on the pixel values. Our dataset will not always contain the exact match, but instead contain different images, different viewing angles of an image, or rotations in the points of interest. Therefore, a more advanced approach is needed to detect the matches between images: feature matching. Image features such as edges and interest point provide rich information on the image content. These features are unique for each image and hence, help in identifying matches between images with similar features. The features of an image will remain, even if there is a change in size, lighting, or orientation, so the approach may prove further useful if the match in the search image is transformed in some fashion. This approach is also more efficient to use if the image has a large resolution. An example of feature matching can be found in figure 3.3.

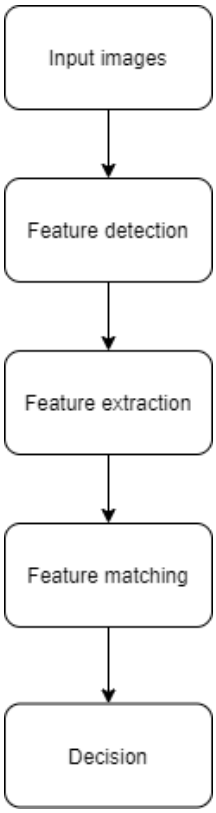


Figure 3.2: Basic pipeline feature matching

In figure 3.2 the basic pipeline for feature based matching can be seen. In the first step, two pictures will be taken as the input for the algorithm. Secondly, the points of interest (keypoints) will be determined. This can be done with three different methods: ORB, SIFT, or SURF. These three methods will be discussed in the sections 3.2.1, 3.2.2, and 3.2.3. In the third phase the descriptors are determined. The information that a descriptor contains, depends on the feature detection method. Descriptors contain the data from the pixels around each interest point. During the fourth step the feature matching will be performed between the descriptors of both images. The features can be matched using Brute-Force or FLANN matching. These methods will be discussed in sections 3.2.4 and 3.2.5. Finally in the decision step, it will be decided whether the pictures form a match. In order to do this, different thresholds need to be determined [25].

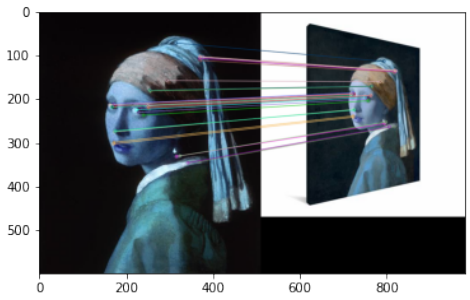


Figure 3.3: Example feature matching

3.2.1. ORB

The first feature detection method that is analysed is the 'Oriented FAST and Rotated Brief' (ORB) method. The ORB method builds on the FAST (Features from Accelerated Segment Test) keypoint detector and the BRIEF (Binary Robust Independent Elementary Features) descriptor [30]. The FAST keypoint detector is a corner detection method, which could be used to extract feature points. The main advantage of FAST is the computational efficiency [31]. BRIEF is a fast and robust local feature detector. It uses simple binary tests between pixels in a smoothed image patch [32] [33]. BRIEF has proven to be robust to lighting, blur, and perspective distortion. However, it is very sensitive to in-plane rotation [30]. In order to solve this, ORB has a rotation matrix which is computed using the orientation of the keypoints and then the BRIEF descriptors are steered according to the orientation.

The ORB method uses a multiscale image pyramid, this pyramid consists of sequences of images all of which are versions of the same image at different resolutions. Each level in the pyramid contains the down-sampled version of the image at the previous level. Once ORB has created a pyramid it uses the FAST algorithm to detect keypoints in the image. By detecting keypoints at each level ORB is effectively locating key points at different scales. After locating keypoints ORB assigns an orientation to each keypoint depending on how the levels of intensity change around that keypoint. For detecting intensity change ORB uses an intensity centroid. The intensity centroid assumes that a corner's intensity is an offset from its center, and this vector may be used to impute an orientation. BRIEF takes all keypoints found by the FAST algorithm and converts them into a binary feature vector so that this vector represents an image (or rather, its keypoints). The binary feature vector, also know as a binary feature descriptor, is a feature vector that only contains ones and zeroes. In BRIEF, each keypoint is described by a feature vector, a bit string of 128 up to 512 bits [30] [34] [35].

3.2.2. SIFT

The second feature based method that will be discussed is SIFT (Scale-invariant feature transform), which was designed by David Lowe to resolve image rotation [36] [34]. The SIFT method for the detection of features in images uses a transformation of the image into a large collection of feature vectors, each of which is invariant to image translation, scaling, rotation, partially invariant to illumination changes, and robust to local geometric distortion [37]. The SIFT algorithm has 4 basic steps:

Firstly, a scale space extrema using the Difference of Gaussian (DoG) is estimated. [1] Secondly, key locations are defined as maxima and minima of the results of differences of Gaussian. Besides this, low-contrast candidate points and edge response points are discarded. Above that, dominant orientations are assigned to the localized keypoints. These steps ensure that the keypoints are more stable for matching and recognition [38]. Thirdly, each keypoint is assigned one or more orientations based on local image gradient directions. This is the key step in achieving invariance to rotation as the keypoint descriptor can be represented relative to this orientation and therefore achieve invariance to image rotation [37]. In the last step the descriptors are computed. The image gradient magnitudes and orientations are sampled around the keypoint location. For this the scale of the keypoint is used in order to select the level of Gaussian blur for the image. In order to achieve the orientation invariance, the coor-

[1] Difference of Gaussians (DoG) is a feature enhancement algorithm that involves the subtraction of one Gaussian blurred version of an original image from another. [36]

dinates of the descriptor and the gradient orientations are rotated relative to the keypoint orientation. The magnitudes are further weighted by a Gaussian function with σ equal to one half the width of the descriptor window [39].

An extra advantage of the SIFT method is the thresholding process. By setting a threshold, all matches whose distance ratio is greater than this threshold value will be rejected. This will eliminate false matches, but possibly also correct matches [37].

3.2.3. SURF

Speed Up Robust Features (SURF) is essentially a speeded-up version of SIFT [40]. The SURF algorithm is based on the same principles and steps as SIFT; but specifics in each step are different [41]. Where SIFT calculates the extrema of the Difference of Gaussians applied to the scale space, SURF approximates this with box filters based on the Hessian matrix (Eq. 3.1) at scale σ. The inputs of the Hessian matrix are the convolutions of the second-order derivative of Gaussian. These inputs are called the Laplacians. Instead of Gaussian averaging the image, squares are used for approximation since the convolution with squares is much faster if the integral image [2] is used. Furthermore, this process can be done in parallel for different scales, making it appealing in terms of compute power utilization [34]. The feature descriptor in SURF uses a BLOB [3] detector which is based on the same Hessian matrix to find the points of interest. The determinant of the Hessian matrix is used as a measure of local change around the point and points are chosen where this determinant is maximal [41]. The descriptor will provide a unique and robust description of an image feature. For example by describing the intensity distribution of the pixels within the neighbourhood of the point of interest. The dimensionality of the descriptor has direct impact on both its computational complexity and point-matching robustness/accuracy. A short descriptor may be more robust against appearance variations, but may not offer sufficient discrimination and thus result in too many false positives [41]. A neighborhood around the key point is selected and divided into subregions and then for each subregion the wavelet [4] responses are taken to get the SURF feature descriptors. A simple rectangular Haar-like feature can be defined as the difference of the sum of pixels of areas inside the rectangle, which can be at any position and scale within the original image [45]. The sign of Laplacian which is already computed in the detection is used for underlying interest points. The sign of the Laplacian distinguishes bright blobs on dark backgrounds from the reverse case. In case of matching, the features are compared only if they have same type of contrast (based on sign) which allows for faster matching [34].

$$H(x, y, \sigma) = \begin{bmatrix} L_{xx}(x, y, \sigma) & L_{xy}(x, y, \sigma) \\ L_{xy}(x, y, \sigma) & L_{yy}(x, y, \sigma) \end{bmatrix} \tag{3.1}$$

3.2.4. Brute-Force

As soon as the features are extracted, the calculated descriptors of each image should be compared for matching. One of the methods for this is Brute-Force (BF) matching. The BF method is straightforward and relies on trying every option rather than advanced techniques [46]. While the Brute-Force search is simple to implement and will always find a solution if it exists, implementation costs are proportional to the number of candidate solutions, thus the more descriptors there are the longer it takes. The main disadvantage of the Brute-Force method is that, for many real-world problems, the number of natural candidates is prohibitively large. This makes it a rather slow method [47].

3.2.5. FLANN

A faster alternative to BF is FLANN (Fast Library for Approximate Nearest Neighbours). This approach offers an approach which significantly speeds up the computation, at the price of only being able to guarantee that the results are approximations. The FLANN matcher will only try a specified number of options, instead if trying all of them [48] [49].

[2] The entry of an integral image $I_{\sum}(x)$ at a location $\mathbf{x} = (x, y)$ represents the sum of all pixels in the input image I of a rectangular region formed by the point $\mathbf{x}$ and the origin, with $I_{\sum}(\mathbf{x}) = \sum_{i=0}^{i \leq x} \sum_{i=0}^{i \leq y} I(i, j)$. With $I_{\sum}(x)$ calculated, it only takes four additions to calculate the sum of the intensities over any upright, rectangular area, independent of its size. [42]

[3] Methods that are aimed at detecting regions in a digital image that differ in properties. [43]

[4] A wavelet is a wave-like oscillation with an amplitude that begins at zero, increases, and then decreases back to zero. [44]

3.2.6. Recap

This section discussed some of the OpenCV feature matching methodologies. For the application of an image search engine, feature matching seems to be the better solution in terms of matching. Still there are 6 possible feature matching combinations that could be implemented. In the remainder of this thesis the choice regarding the best combination will be based on the program of requirements 2, research into their performance, and the results from testing 5.

According to research done by E. Karami, S Prasad, and M. Shehata; ORB is the fastest algorithm while SIFT performs the best in most scenarios. The results are based on testing the algorithms on images with varying intensity, rotation, scaling, fish eye distortion, and the addition of noise [34]. As it was stated in the program of requirements that performance is preferred over speed, SIFT seems to better suit our application. The higher performance of SIFT is mainly due to the filtering of qualitative matches. The performance of SURF comes close to that of SIFT but performs significantly worse on noisy images [34] [50].

Secondly, a decision on the matcher has to be made. This decision is based on the same trade-off requirement. As the best performance is preferred, it is clear that the BF matcher will be best suited for this application. Therefore the decision is made to continue with the SIFT-BF method, as this should result in the best performance.

In order to validate this hypothesis, tests will be performed on 3 different combinations. These tests will focus on ORB extraction with the Brute-Force matcher and the SIFT extractor with both matchers. SURF is disregarded as this does not offer any advantages according to the program of requirements for this application. AS ORB is expected to have a lower performance, it will not be tested in combination with the FLANN matcher.

4

Design Process

Deciding on an algorithm to be used can be quite a challenge. The current state of the industry is that the most modern solutions outperform older technologies by a long shot. It follows that even though the newer technologies potentially perform better, code implementations of these algorithms are sparsely available, still in the proof of concept phase, or not available all together. Therefore it was decided to not implement these new algorithms from scratch, but to use readily available and tested libraries to bring this project to a timely end. The analysis in chapter 3 has shown some of these available OpenCV methods.

In this chapter the process of designing the algorithm for the search engine is described. First a description will be given into how images can be prepared so that they can be optimally searched. Followed by this, some different implementations of the methods mentioned in chapter 3 are discussed. Each implementation will be discussed using snippets. The full code of the final implementations can be found in Appendix H. The results of these implementations will be discussed in chapter 5. Besides this, this chapter covers the indexing of the database and multithreading.

4.1. Pre-Processing

Images in a real-world databases will almost always vary in size and resolution. The search engine designed in this thesis should be able to process all of these images. Historic images often have a very high resolution to retain as much detail as possible and because they are often made in a professional setting. Take for example a universities collection of maps, these images will be in a very high resolution to allow scholars to study the map thoroughly.

The SIFT algorithm takes a very long time to extract features from High resolution images, and thus requires a relatively large amount of memory in the process. For this reason all ”large” images which are contained in the search database are downscaled before features are extracted. Code was written which takes an image and downscales it so that the largest axis has no more than 3000 pixels while retaining the aspect ratio of the image. If for example an image was provided of 5000x2000 pixels, it will be downscaled to 3000x1200 pixels. This code can be found in appendix B.

4.2. Feature Extraction

This section will explain the implementation of a SIFT and ORB feature extractor. First, the ORB implementation will be discussed since this is the more basic version. Afterwards, the feature extraction using SIFT will be explained.

ORB

(part of) The ORB feature extraction code is show below.

Listing 4.1: ORB feature extraction

```
orb = cv2.ORB_create()
```

```
keypoints1, descriptors1 = orb.detectAndCompute(img1, None)
keypoints2, descriptors2 = orb.detectAndCompute(img2, None)
```

In the first line of code the ORB detector object is created. The function 'orb.detectAndCompute' will detect the keypoints and calculate the descriptors around these point of interests. This function takes two input parameters. The first input will be the image and the second parameter is the mask. The mask parameter can specify where to look for keypoints [51]. This is set to none, since this is image dependent. The resulting keypoint array will consist of the following information:

- Coordinates of the keypoints.
- Diameter of the meaningful keypoint neighborhood.
- Computed orientation of the keypoint (-1 if not applicable); it's in [0,360) degrees and measured relative to image coordinate system (y-axis is directed downward), i.e in clockwise.
- The response by which the most strong keypoints have been selected. Can be used for further sorting or subsampling.
- Octave (pyramid layer) from which the keypoint has been extracted.
- Object class (if the keypoints need to be clustered by an object they belong to).

The descriptor array consists of the computed descriptors. Each descriptor is a 32-element vector making the total size of the descriptor array $keypoints \cdot 32$ [52] [53].

SIFT
The code for a SIFT feature extractor looks very similar to that of ORB. In the first step a SIFT object is created. Followed by the search for keypoints and computation of the corresponding descriptors, using the same detectAndCompute function. The keypoints that result from this function for SIFT contain the same kind of information as for ORB. With SIFT however, the descriptors are a 128-element vector, this results in a descriptor matrix of $keypoints \cdot 128$. The code is listed below.

Listing 4.2: SIFT feature extraction

```
sift = cv2.SIFT_create()

keypoints1, descriptors1 = sift.detectAndCompute(img1, None)
keypoints2, descriptors2 = sift.detectAndCompute(img2, None)
```

4.3. Feature Matching

As shortly described in chapter 3 there are two ways of matching: BF and FLANN. Firstly, the implementation of BF matching is discussed, followed by the implementation of FLANN.

Brute-Force
The code snippet for the Brute-Force matcher can be found below.

Listing 4.3: BF feature matcher

```
bf = cv2.BFMatcher()
matches = bf.Match(descriptors1, descriptors2)
```

The first step in Brute-Force matching is creating the Brute-Force object 'bf'. The function 'cv2.BFMatcher' consists of two optional parameters. The first one is the 'normType' that specifies the distance between features as a measurement of similarity. By default this is set to 'cv.NORM_L2' which is suited for SIFT and SURF. For binary string based descriptors like ORB 'cv.NORM_HAMMING' should be used. The Hamming distance is defined as the number of bit positions that are different between bit strings of equal lengths. The second parameter is the Boolean variable 'crossCheck', which is false by default. If it is true, the matcher returns only those matches with value (i, j) such that the i-th descriptor in set

A (descriptor image 1) has the j-th descriptor in set B (descriptor image 2) as the best match and vice-versa. That means that the two features in both sets should match each other. That is, the two features in both sets should match each other. It provides consistent result [52] [53].

Once the object is created there are two different methods for matching. These are 'BFMatcher.Match()' and 'BFMatcher.knnMatch()'. The first one returns the best match, whereas the second method returns 'k' best matches. The 'k' can be specified as an input parameter. For this implementation only the best match is used, and therefore 'BFMatcher.Match()' is used in this code snippet [52] [53]. A sample of the

FLANN
The code snippet for the FLANN matcher can be found below.

Listing 4.4: FLANN feature matcher

```
FLAN_INDEX_KDTREE = 0
index_params = dict(algorithm = FLAN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)

flann = cv.FlannBasedMatcher(index_params, search_params)

matches = flann.knnMatch (descriptors1, descriptors2, k=2)
```

Before being able to create the FLANN object, two dictionaries that specify the algorithm to be used have to be passed on.

The first one is the 'index_params' dictionary. This will create a dictionary algorithm by passing the 'FLAN_INDEX_KDTREE' into 'index_params. The second input to the 'index_params' dictionary is the number of trees. For a SIFT or SURF implementation this is equal to 5 trees. The 'index_params' dictionary looks a bit different for ORB, as this is binary based [52] [53]. It should be defined as follows:

Listing 4.5: FLANN feature matcher for ORB

```
index_params= dict(algorithm = FLANN_INDEX_KDTREE,
                   table_number = 6, # 12
                   key_size = 12,     # 20
                   multi_probe_level = 1) #2
```

Secondly the 'SearchParams' dictionary is specified. This specifies the number of times the trees in the index should be recursively traversed. Higher values will result in a better precision, but also take more time to compute. After creating the FLANN object using the 'FlannBasedMatcher' function, the matches can be calculated using 'flann.knnMatch', which will calculate the 'k' nearest neighbour matches [52] [53].

4.4. Decision Making

After retrieving the data of the matches as explained in the sections before, it is time to make a decision whether the two input images can be regarded as a match or not. The processing of the array of matches differ between ORB and SIFT. This section will explain the decision making part of the code for both implementations. An important note is that for the decision matching methodology does not matter.

ORB
The processing of the ORB matches is quite basic. Since ORB does not offer any possibility of filtering the quality of matching, the array of matches will just be sorted from low to high based on the distance that the matcher gave to the match. The code will look as listed below.

Listing 4.6: ORB decision

```
matches = sorted(matches, key=lambda x:x.distance)
```

```
distant = []
for i in range(30):
        single = matches[i]
        distant[i] = single.distance
```

The sorted function will sort the matches from low to high. This function has one required parameter, which is the matches. Furthermore, it has two optional parameters: reverse and key. The reverse input can either be true or false (default). If it is set to true the sorted list is reversed. The key input will serve as a key for the sort comparison. For this application the key is set to 'lambda x:x.distance'. This will return the sorted list containing the items from the matches [52]. After sorting the matches, an array containing the distances of the best 30 matches is created. Using the distances from this array, a decision can be made whether 2 images match or not.

SIFT
Processing the matches from SIFT is somewhat more advanced, as it has the possibility of validating the quality of the matches. This can be done using David Lowe's ratio test [36]. The code for this looks as follows:

Listing 4.7: SIFT decision

```
good = []

for m,n in matches:
        if m.distance < 0.6*n.distance:
            good.append([m])
```

In this for-loop all matches with a distance ratio greater then 0.6 are rejected. The 0.6 was arbitrarily chosen by Lowe [51] [37]. To decide whether a combination of two images match or not, a percentage of the number of qualitative matches is used. This is calculated as follows.

Listing 4.8: Decision percentage

```
perc = len(good)/len(matches)
```

This means that in order to decide whether a set of two images form a match there are three threshold values to adjust:

1. The distance ratio
2. A minimum percentage of qualitative matches 'perc'
3. A minimum number of qualitative matches 'len(good)'

The threshold values can be chosen arbitrarily, but for this project it has been decided to perform a Monte Carlos analysis to determine the best values. The decision of these values have a huge impact on the results of the algorithm. In order to decide which values are the best, three performance variables are calculated for each possible threshold combination: balanced accuracy, precision, and recall. The algorithm should provide an option which is ideal for all three parameters. The code written for this can be found in the appendix C. The code takes as an input a CSV file containing arrays with the distance ratio's between two images and a label whether it should be a match or not. The function will analyze the balanced accuracy, precision, and recall for all possible thresholds. For each percentage to decide whether two images match, the function will search for optimum values for balanced accuracy, precision, and recall. This is done by looping through a minimum number of qualitative matches and the decision threshold that decides which ratio's are qualified as a good match.

4.5. Indexing the Search Database

As the extraction of features is a rather slow process [34], an index can be created to store the keypoints and descriptor of each image beforehand. Due to this, features have to be extracted only once which will speed up the process extensively.

A hurdle is the saving of the datatype in which the feature descriptors of the image are stored. In the used code the feature descriptors are stored as numpy matrices, but in a simple file format you want to store an ASCII representation of this data. It is also undesirable to store the data in a format which requires more compute power to convert it back into a numpy array. It was therefore chosen to use the Python *Pickle* library to convert the numpy format into ASCII which can be sorted in a CSV file. Code for this serialization and deserialization can be found in appendix D.

4.6. Multithreading

Modern CPU's have multiple CPU cores in them, but simple programs only use a single core by default. To make the search engine faster, it should use all the cores available. A program can run code on multiple cores by splitting itself up into multiple "threads". Using multiple threads has the benefit of running parts of the code consecutively, thus increasing speed. Every thread is a child of the original program, and every child thread gets its jobs from the main thread. For the purposes of this thesis the Python library *concurrent.futures.ThreadPoolExecutor* was used. This library allows the program to queue up a large amount of work, in which the threads take a job when they are finished with their current one. The 'ThreadPoolExecutor' puts all *worker threads* in a pool, which are now ready to accept jobs. The main thread will schedule all images to be looked through in a queue, the worker threads take a job from the queue when they are done finishing their previous image. This is done until the queue is cleared. To queue all the jobs and start all the threads the following code is used:

```
import concurrent.futures

with concurrent.futures.ThreadPoolExecutor(max_workers=10) as executor:
    executor.map(thread_function, jobs)
```

In which the $thread\_function$ is the function which the thread will execute, and $jobs$ is a list of jobs for the thread function to execute. The parameter $max\_workers$ defines the amount of (worker)threads, which is chosen to be 1.5 times the amount of cores available.

An issue in multithreaded programs is the use of shared resources. A thread needs a place to write its results to, and other threads should not write to the same resource at the same time to prevent race conditions. Therefore the principle of *locks* is used. When a thread wants to output its results to a shared resource it acquires the lock and releases the lock when its done with the resource. When a different thread tries to do the same it will see the lock is used and wait for it to be released by the original thread, thus preventing race conditions.

In the case of the search engine implementation several CSV need to be written by different threads. The code for only allowing a single thread to write at the same time is as simple as:

```
import threading

write_lock = threading.Lock()
def write_row(data):
  logging.debug('Waiting for csv lock')
  with write_lock:
    logging.debug('Writing csv')
    csvWriter.writerow(data)
    logging.debug('csv write done')
```

5

Testing and Results

To check the functionality of the implemented combinations, it is important to run multiple tests. For the tests, two datasets were created. The first dataset contains a small set of images which are mostly meant to test basic functionality of the algorithms. The 3 different implementations will be tested on this dataset. The best performing algorithm will be optimised for the second dataset. The second dataset is meant to test very difficult cases, but also to test on a wide variety of images. Before discussing these tests, this chapter will discuss the used hardware. This chapter is concluded with a comparison with the results of the *Deep Learning* approach

5.1. Compute Hardware

The systems developed in this thesis do require quite the compute power to search through datasets. Especially unoptimised feature extracting (SIFT and SURF) takes a very long time to run. Optimisations such as indexing and downscaling have been developed to make the runtimes within boundaries. Further optimisations definitely need to be taken, which is also addressed in the discussion 6 of this thesis.

The authors of this thesis originally used Google Collab to collaborate on code and have a free compute instance to run CPU and GPU jobs. Google Collab is good for in initial prototypes, but as soon as work needs to be done by multiple people at the same time the caveats quickly become apparent. Google Collab does not support multiple people editing the notebook at the same time. An alternative is Deepnote, which gave the authors more flexibility and collaboration tools. The results that will be presented in this chapter, have been run on the Deepnote server.

5.2. Dataset Tests

This section will discuss the results of the performed tests on the datasets. Both datasets can be found at: [54].

5.2.1. Basic Functionalities Test

The first dataset consists of 6 needles that are searched for in a haystack of 27 images. Each needle image has 3 matching images in the haystack. One of these 3 matches is chosen extra difficult. The different needles are 6 different historical artifacts that have different features. On this dataset three methods will be tested and evaluated. The methods that are evaluated are ORB-BF, SIFT-BF, and SIFT-FLANN.

The decision whether the ORB-BF combination results in a match will be based on the mean of the first 30 matches. If the mean is below 40, the algorithm will match the images. The threshold values for both SIFT implementations have been set equally. The distance ratio threshold is set to 0.6 (just as Lowe did [36]). The percentage of qualitative matches should be at least 1% in order to match. For this case no minimum number of matches is specified.

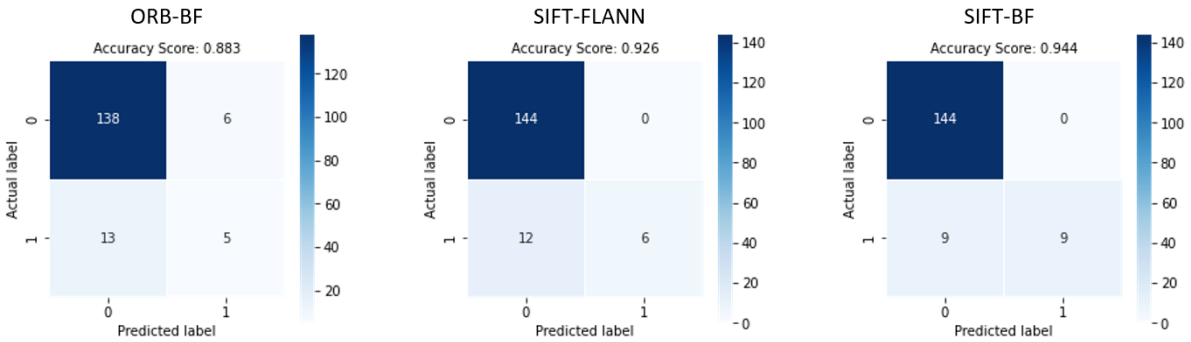


Figure 5.1: Confusion matrices of first dataset

The results of each method can be seen in figure 5.1 and table 5.1. Form the results it can be concluded that ORB-BF has the worst performance in terms of precision, recall, and balanced accuracy. The main cause of this is the lack of filtering non-qualitative matches, as the algorithm always bases its score on the 30 best matches. This may not always result in a qualitative decision. On the other hand the ORB-BF is much faster than the other two methods. Both SIFT methods score maximal on precision. Due to the filtering of matches, the algorithm returns no wrong image-matches. The BF matcher has a higher recall and therefore higher balanced accuracy. This is due to the fact that the BF matcher checks all options, and FLANN does not. This has resulted in a small time difference. The times are measured of the run on the Deepnote server. As the dataset was stored in Google Drive, an unknown portion of this time goes to retrieving the image from google Drive. As according to the program of requirements performance goes over time. The SIFT-BF method meets the balanced accuracy requirement while the SIFT-FLANN method does not. The conclusion can be made that SIFT-BF is the best method for this application.

	Precision	**Recall**	**Balanced Accuracy**	**Extraction time (s)**	**Matching time (s)**	**Total time (s)**
ORB-BF	0.45	0.28	0.62	0.10	0.01	0.12
SIFT-BF	1.00	0.50	0.75	1.69	1.94	3.62
SIFT-FLANN	1.00	0.33	0.67	1.72	1.17	2.89

Table 5.1: Results of tests on dataset 1 using different methods with default thresholds

5.2.2. Complete Functionalities Test

The second data set was created to test very difficult use cases, but also to test on a wide variety of images. The data set contains 184 images and consists of low and high resolution images, detailed and undetailed images, blurred images, and images with different lighting conditions. On top of that, this dataset contains a number of random images from Wikipedia to check the functionality on those images. This results in 25140 unique comparison combinations. In order to achieve the best possible functionality of the algorithm, the thresholds will be optimized using this dataset. The threshold code as explained in section 4.4 will be used for this. As an input a comma-separated values file (CSV) containing all distance ratios from each combination from the second data set is taken. Running the code has resulted in values for an optimal balanced accuracy, precision, and recall. The results are presented in figure 5.2 and table 5.2.

In the table 5.2 the results of the optimization are shown. The table contains the precision, recall, balanced accuracy, maximum value for the distance ratio, minimum number of qualitative matches, and the minimum percentage of qualitative matches. Each row contains the values for the optimized results of one parameter. The values for a maximal precision seem to be the best values. This row is the only one containing values that meet the requirements stated in the program of requirements 2. Therefore the algorithm will be implemented with these threshold values.

According to these results the recall is still quite low (35%). As can be seen, a recall of 1 is possible, but

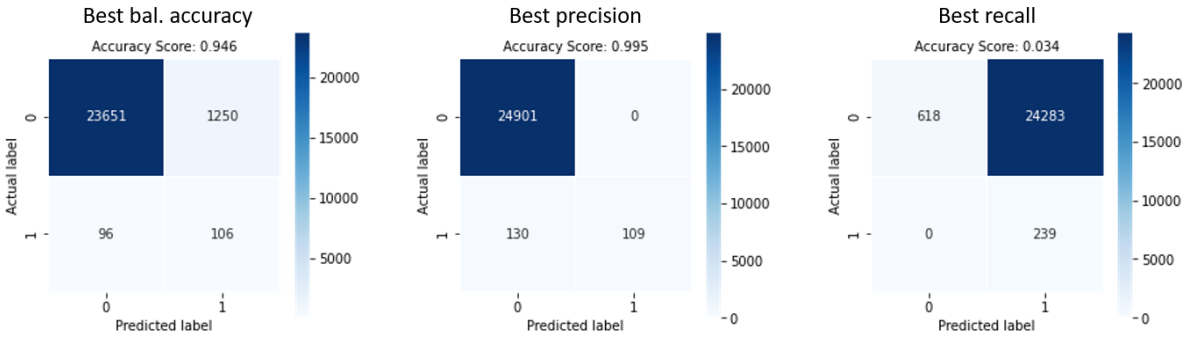


Figure 5.2: Confusion matrices of second dataset after optimising

	Precision	Recall	Balanced Accuracy	Ratio Threshold	Minimum Matches	Min %
Best Balance	0.078	0.523	0.737	0.6594557239209874	5	0.009
Best Precision	1.00	0.352	0.728	0.5519978894151195	21	0.037
Best Recall	0.010	1.00	0.512	0.8036316693804482	2	0.003

Table 5.2: Results threshold optimization

this will influence the precision drastically. The fact that not all matches are found lies in the threshold values. High quality pictures have a high number keypoints, whereas low quality picture have a small amount of keypoints. If these two images should match, the minimum percentage of qualitative matches is not reached due to the large difference in keypoint amounts. Lowering this percentage makes it easier for false matches to be matched as well, as they also score a very low percentage of qualitative matches.

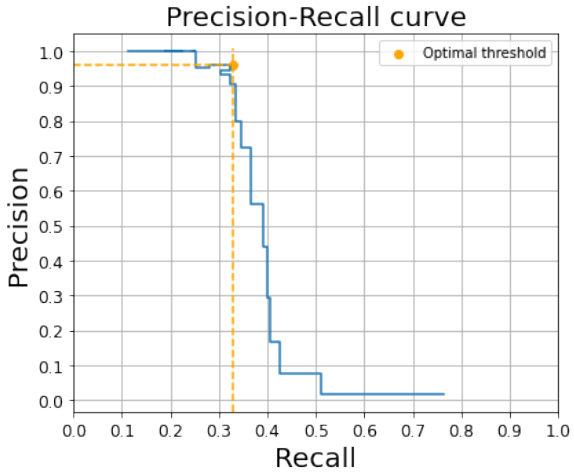
An analysis was also done how the precision-recall curves and the balanced accuracy is influenced by changing the threshold parameters. Precision-recall curves show the trade off between the fraction of relevant instances among the retrieved instances (precision) and the fraction of relevant instances that were found (recall) for a varying parameter. Thus it can for example be shown that when a parameter is chosen too high the precision will increase, but the recall will worsen.

With the already calculated optimum values for the three parameters, the precision-recall curves and the balanced accuracy are calculated one by one by varying a single parameter and keeping the other parameters in their optimum. This gives an indication what the effect is of varying every parameter individually. The precision-recall curves can be found in figure 5.3. From the Precision-recall curves it can be seen that the optimal values calculated are indeed a point on the curves in which the precision is near 1 and the recall is thus slightly lower. It is useful to note that if the requirement of a precision of 1 is even slightly lowered, the recall rate increases greatly. For example in figure 5.3a the requirement of a precision of 1 does mean that quite some images are missed, the recall rate is only 25%, but if the precision is 80% the recall rate jumps up to 35%. For the purposes of a search engine however, a precision close to 1 is desirable to ensure that the results are useful to the user.

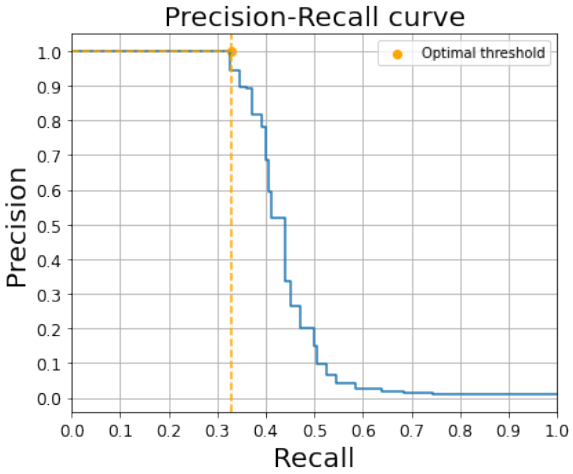
Furthermore the relation between varying every parameter and the balanced accuracy is analysed. The balanced accuracy takes the amount of false positives and the amount of false negatives into account, and can be seen as a performance metric for the database. For every subfigure in figure 5.4 a parameter is varied and the resulting balanced accuracy is plotted. The parameter ”Minimum amount of matches needed” does not influence the balanced accuracy much, but an optimum was found at 21. The other two parameters: ”Keypoint threshold” and ”Match ratio threshold”, do have significant effect on the balanced accuracy.

5.3. Wikipedia indexing

To test the indexing functionalities, an index was created from Wikipedia images (Wikimedia). Wikipedia provides a free to use API with an endpoint to query all images of Wikipedia [55]. An implementation

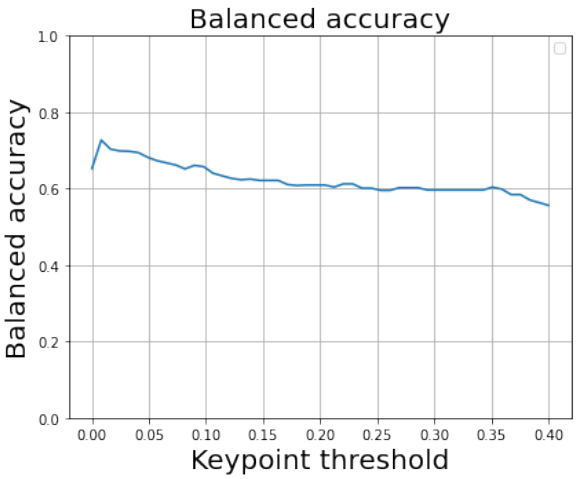


(a) Varying the percentage of total matches needed for an image to be flagged as a match

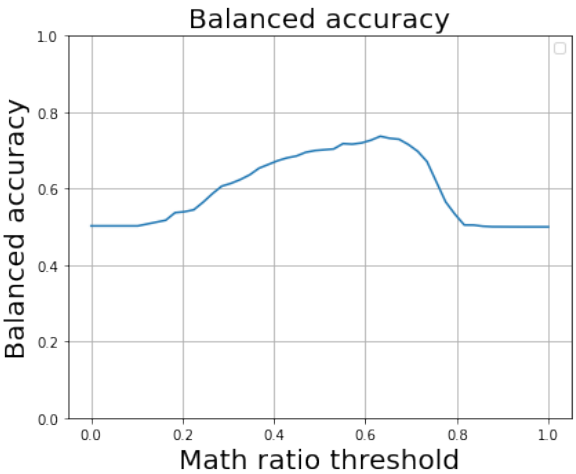


(b) Varying the threshold which decides when a feature combination will be considered as a match

Figure 5.3: Precision-Recall curves for varying parameters

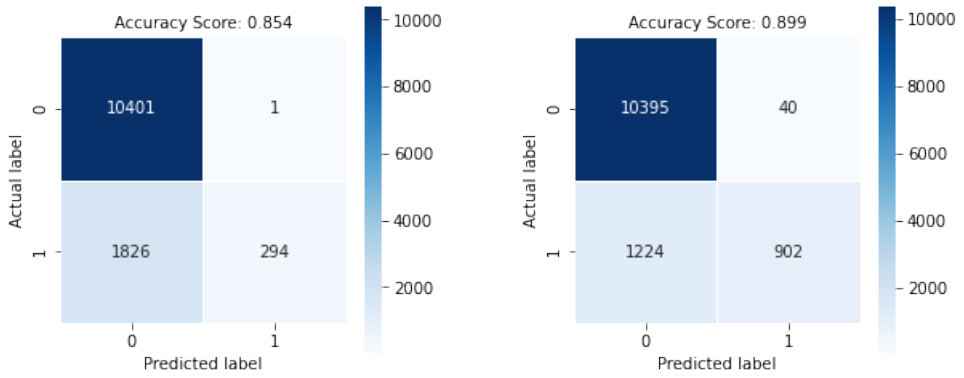


(a) Varying the percentage of total matches needed for an image to be flagged as a match



(b) Varying the threshold when a feature combination is considered a match

Figure 5.4: Balanced accuracies for varying parameters



(a) *Standard* approach (b) *Deep Learning*

Figure 5.5: Confusion matrices comparison

was written which downloads images obtained from the API, extracts the features of the downloaded images, and saves the descriptors required for later matching in a CSV file as explained in 4.5.

The indexing itself resulted in a great reduction in search time. To search for an image in the CSV file the steps required are as follows:

- Extract features of the query image
- For every image in the index load their features (descriptors) and execute the Brute-Force matcher to find matches between the features of the query image and the database image
- Sort all the images which are a match by how good a match they are.

A sample index was created containing over 40.000 images from Wikipedia, which resulted in a database size of ~20GB. This brings the average storage required per image to an average of 0.5 MB. A small sample of the indexed images can be found in appendix E.

Searching through an indexed database thus requires only a single feature extraction run. In the case of the Wikipedia index a test was run using a query image 300x167 pixels. This image took a total of 490 seconds to look through the full index of the 40.000 images in the set. This brings the average time it takes to match to 12 ms per image. Code for searching through and generating the index can be found in appendix F

5.4. Result Comparison Deep Learning

This thesis *Image Search Engine for Digital History: Standard Approach* is part of the project suggested by J. Dauwels. As part of this project, two subgroups were created: *Standard* approach and *Deep Learning*. Due to the fact that the research presented in this (and the *Deep Learning*) thesis will be used to further the *Engineering Historical Memory* project, it would be useful for EHM if the results of these two approaches are comparable. For the comparison a dataset consisting of 159 images was created. These images are related to historical artifacts, but the images chosen are somewhat more difficult then the second dataset (section 5.2.2). It consists of more images with higher quality and detail. A link to the dataset can be found here: [54]. The *Standard* approach algorithm uses the optimised thresholds from section 5.2.2.

	Bal. accuracy	Precision	Recall	Total time (s)
Standard approach	0.569	0.997	0.139	3.62
Deep Learning	0.710	0.958	0.424	36.2

Table 5.3: Results comparison

The results can be seen in figure 5.5 and table 5.3. From the result two things can be concluded: (1)

the accuracy of *Deep Learning* is better than that of the *Standard* approach, (2) the *Deep Learning* algorithm takes considerably longer to match a single image. The lower recall is mainly caused by the lower performance of the *Standard* approach algorithm. As explained in section 5.2.2 the algorithm has trouble with matching images with high quality to an image with relatively low quality. The *Deep Learning* approach scores significantly better due to it's ability to better match such images. On the other hand, comparing the total times shows that the *Deep Learning* algorithm is a rather slow method. An important note is that the times are for 1 CPU core. If multithreading is used, both times will be sped up extensively. As the program of requirements of this thesis states, accuracy is more important than the speed at which images are searched for. However seeing as this research is part of a bigger project commissioned by EHM, EHM might consider having time constraints weigh more in their Program of Requirements.

6

Discussion and Conclusion

This thesis has discussed the implementation and working of different feature matching methods. After extensively analysing these methods, SIFT-BF performed best for this application. The main advantage of the SIFT-BF implementation is the extremely high precision of 1, due to which it can be guaranteed that the algorithm will give qualitative results if a match is found. Still the recall was quite low (35%), even though this value meets the program of requirements, this may be improved to achieve a better performance.

If the requirements of chapter 2 are evaluated, the SIFT-BF method has proven to meet all of these. The results that are presented in chapter 5 show that a precision of 1.0, a recall of 0.35 and a balanced accuracy of 0.73 can be achieved by optimizing the parameters. Besides this, the code is written in python in order to connect with the existing code of EHM. On top of that SIFT-BF can be implemented with OpenCV and is therefore free for academic use. The second dataset consisted of images larger then 10 MB, which resulted in the high number of keypoints. Due to the resizing images bigger then 10 MB can also be processed.

6.1. Further Work

In order to improve the functionality of the algorithm, some other options can be analysed. For example analyzing the possibility of implementing the code in C using the OpenCV library in C in order to speed up. As OpenCV codes are original written for C, and will therefore work faster in a C environment. Another advantage of C might be the possibility of using GPU support, which can speeds up the functionality enormously. The Python version of OpenCV does not support some required GPU functionality, but the C implementation does.

Secondly, to minimize the storage for indexing, it could be possible to store the keypoints and compute the descriptors instead of saving the descriptors. Also the efficiency in storing the data should be further researched. Ideally a format is required which takes up the least amnount of space, but which can be decoded the fastest.

Also this thesis has not researched the disadvantages of down-scaling large images, as no large computing power was available. For new research it might be interesting to research the effects of down-scaling and decide whether the obtained speed up is worth the loss in details.

Finally, more research can be done in the decision making for matching. In this thesis the difference in number of keypoints between two images is disregarded. This may influence the resulting percentage of qualitative matching. For example if image 1 has an extreme high quality with 20.0000 keypoints and image 2 is of low quality and only has 80 keypoints, it is impossible to achieve a reasonable percentage of qualitative matches. New research could dive into other ways of matching to also retrieve these edgecases. At the moment of writing, not much research has put into this.

A

Examples of Algorithms

A.1. Template Matching

A.1.1. Formulae

The formulae for the different comparison methods within template matching, as described in chapter 3, can be seen in equations A.1 through A.6, where equations A.7 and A.8 explain the variables present in equations A.5 and A.6.

$$R(x,y) = \sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2 \tag{A.1}$$

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}} \tag{A.2}$$

$$R(x,y) = \sum_{x',y'} (T(x',y') \cdot I(x+x',y+y')) \tag{A.3}$$

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') \cdot I(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}} \tag{A.4}$$

$$R(x,y) = \sum_{x',y'} (T'(x',y') \cdot I'(x+x',y+y')) \tag{A.5}$$

$$R(x,y) = \frac{\sum_{x',y'} (T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}} \tag{A.6}$$

$$T'(x',y') = T(x',y') - 1/(w \cdot h) \cdot \sum_{x'',y''} T(x'',y'') \tag{A.7}$$

$$I'(x+x',y+y') = I(x+x',y+y') - 1/(w \cdot h) \cdot \sum_{x'',y''} I(x+x'',y+y'') \tag{A.8}$$


Figure A.1: A picture of Max Verstappen and a smaller template of his helmet [56]

A.1.2. Example

In this appendix an example is shown with template matching an image of Max Verstappen's helmet within the original image A.1. The template matching algorithm implemented in OpenCV return the pictures in grayscale with a box around the best match (this box is the same size as the template image). As can be seen in the images below A.2 through A.7, not all comparison implementations return the same and/or correct result. Repeating this example with a picture of Messi as stated in the example [29] also shows that not all implementations return the same and/or correct result.

A.2. SIFT Feature Extraction

In this example we will show how the SIFT algorithm extracts and defines features in an image according to [51]. As input image we will be using a picture of Albert Einstein A.8. A visual interpretation of the steps described below can be seen in images A.9 through A.16.

1. Doubles the width and height of the input image using bilinear interpolation [57]
2. The image is then blurred using Gaussian convolution [58]
3. Further convolutions are applied to the image in 5 steps with increasing standard deviation, these 6 images together are then called an octave
4. The antepenultimate (second from last) image in the convolution 'sequence' is downsampled and starts a new octave
5. Repeat steps 3 and 4 until the antepenultimate can no longer be downsampled, we have created a scale space
6. For each pair of horizontal adjacent images it calculates the difference of Gaussians [59] for the individual pixels (we go from 6 images per octave to 5)
7. Pixel values are compared with its 26 neighbor pixels (8 pixels directly around it in the same image and 9 pixels in the images left and right) and local maxima and minima are detected (we go from 5 images per octave to 3
8. Local minima or maxima with a low absolute value are discarded
9. ??? Deze is nog vaag ??? Discrete coordinates of the extrema are refined by approximating the quadratic Taylor expansion of the scale space function [1] and calculating its extrema
10. Remove extrema which lie on edges by comparing the principal curvatures [60] of the scale space function at the corresponding location
11. Assign each remaining point its reference orientation by approximating the gradient, using finite differences [61], of each pixel in a square patch around the point
    - Points without enough neighboring pixels or without a dominating orientation are discarded
12. Calculate the gradient distribution of neighboring pixels to the point, this time with a circular patch and the coordinate system rotated to match the previously obtained reference orientation

[1] Scale space function is seeing an entire octave as a continuous space with three dimensions: the x and y coordinates of the pixels and the standard deviation of the convolution

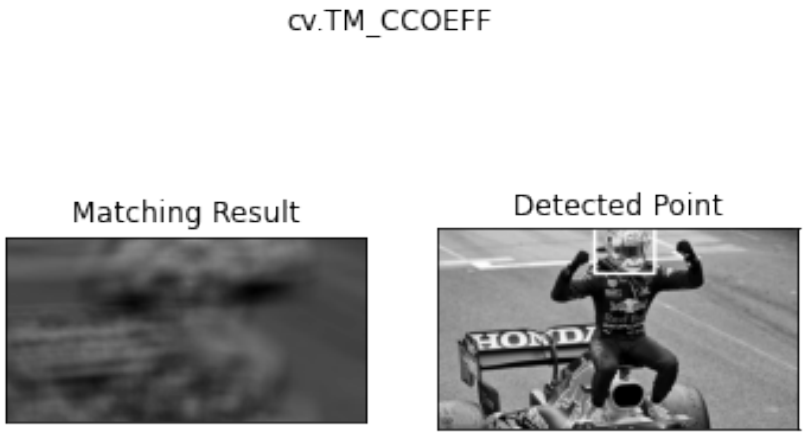


Figure A.2: CCOEFF

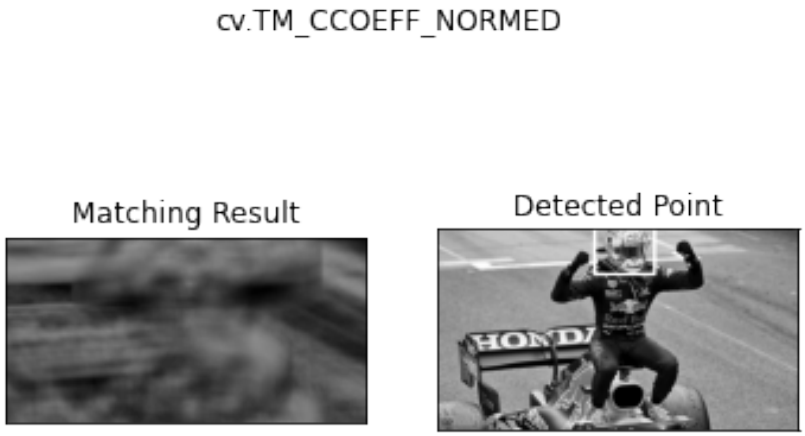


Figure A.3: CCOEFF_NORMED

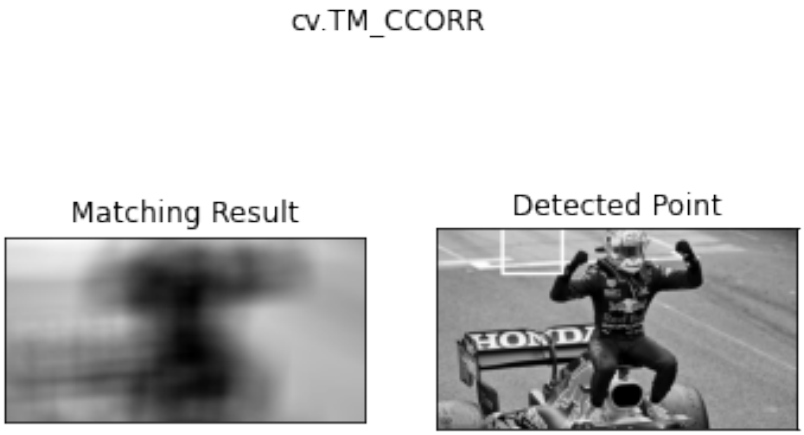


Figure A.4: CCORR

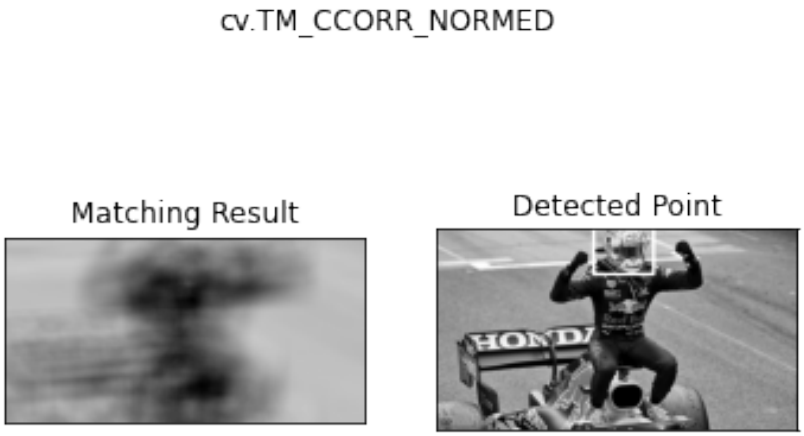


Figure A.5: CCORR_NORMED

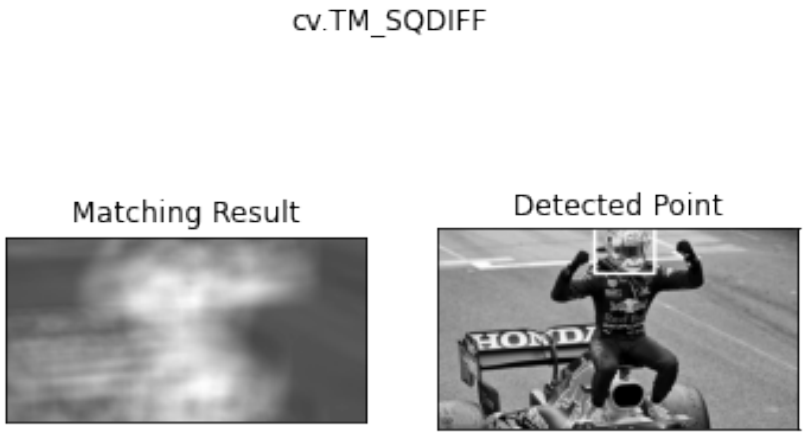


Figure A.6: SQDIFF

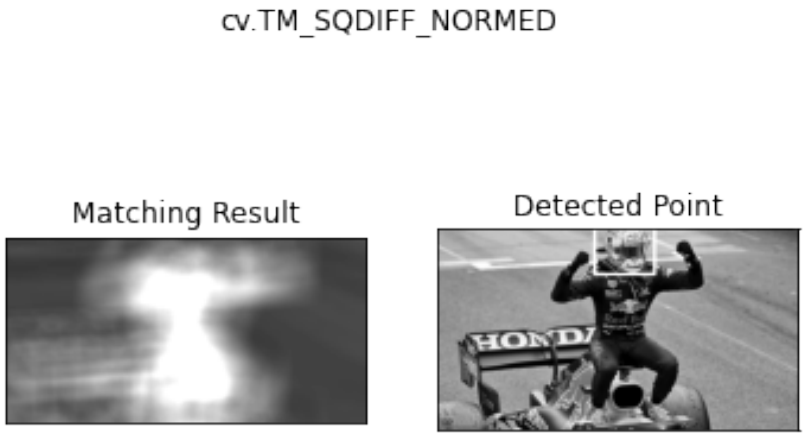


Figure A.7: SQDIFF_NORMED

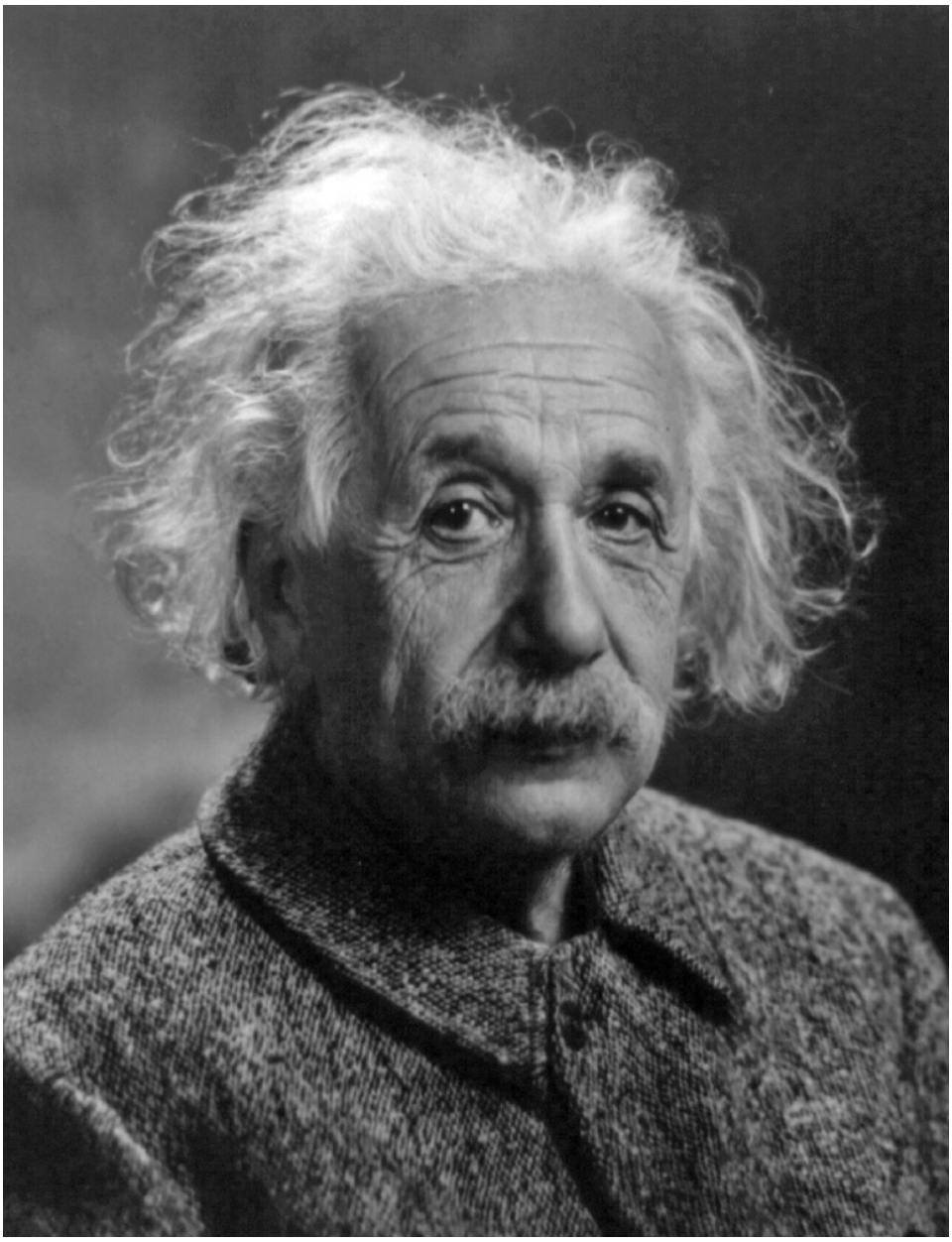


Figure A.8: Albert Eingstein [62]

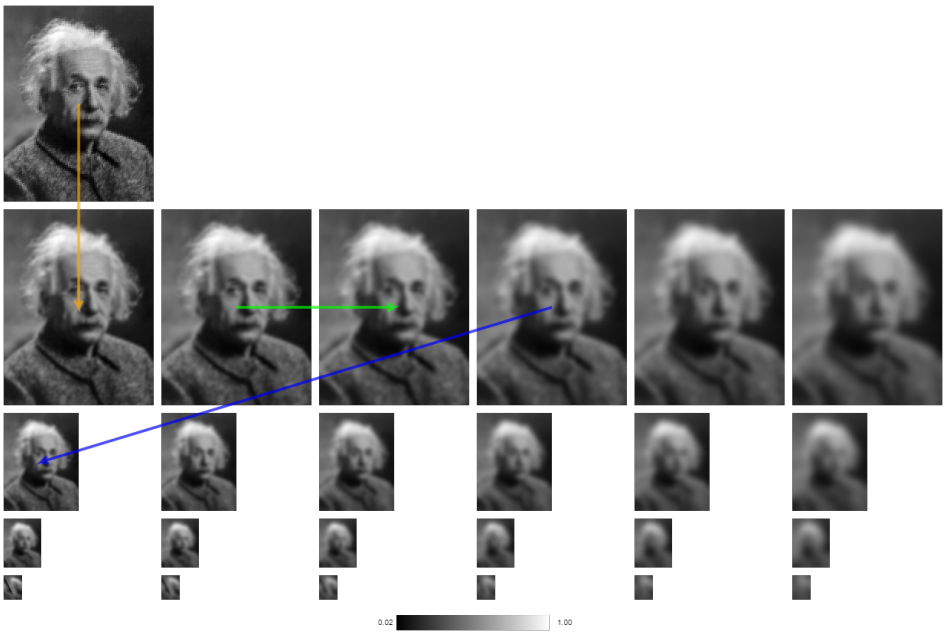


Figure A.9: Steps 1 through 5

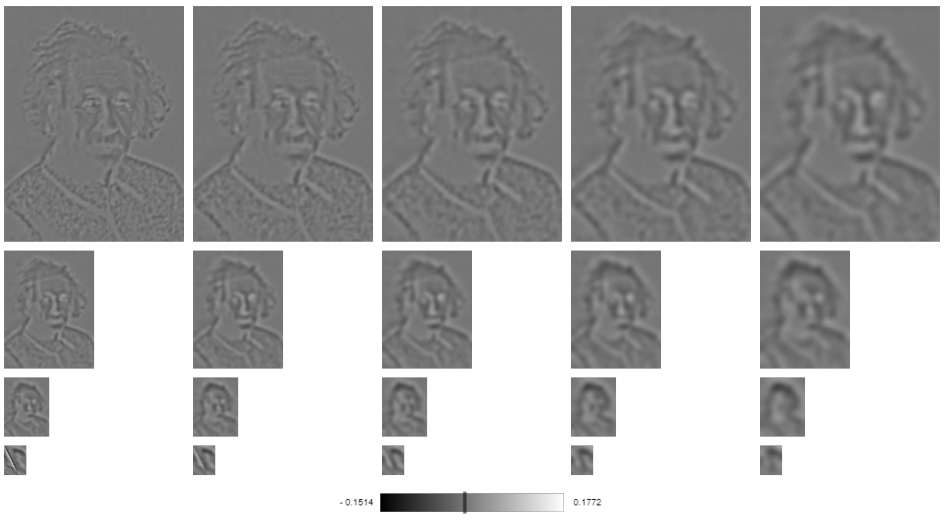


Figure A.10: Step 6

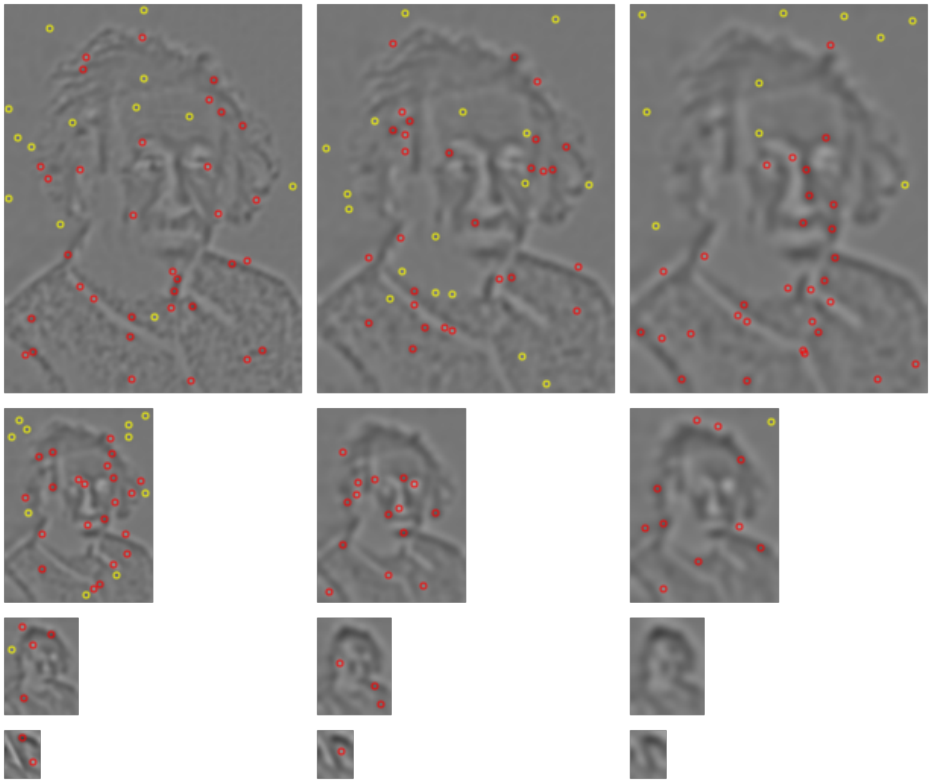


Figure A.11: Step 7

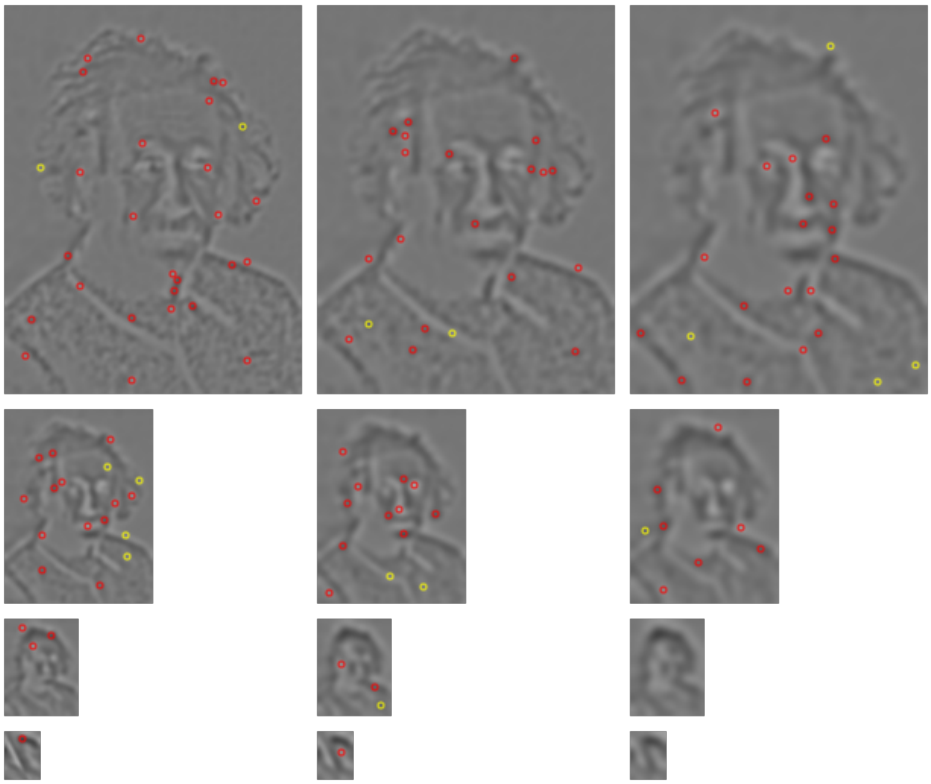


Figure A.12: Step 8

Figure A.13: Step 9 and 10

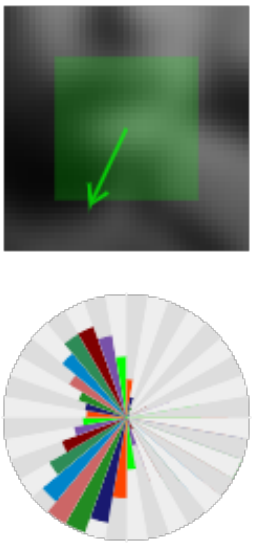


Figure A.14: example of a reference orientation and the corresponding histogram descriptor

Figure A.15: Step 12

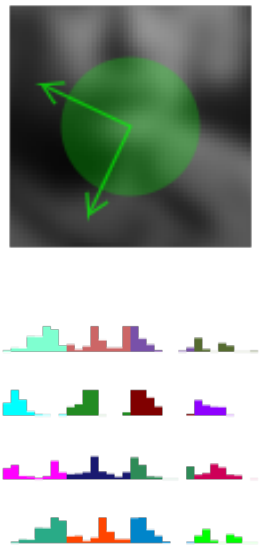


Figure A.16: Result/descriptor of circular gradient description

The descriptor acquired after step 12 is normalized (invariant to global hue changes), calculated relative to the reference orientation (robust against rotation), calculated at different scales and blur levels (invariant to scaling), and relatively immune to noise due to the discarding of certain points during the steps.

B

Image Downscaling Function

Listing B.1: Function which downscales an image if it has a dimension bigger than 3000 pixels

```
import cv2 as cv

# Scales an image down to fit in the bounding maxWidth and maxHeight
# Image should be OpenCV image object
def boundFit(img, maxWidth=3000, maxHeight=3000):
    # If already fitting in side max
    if (img.shape[0] < maxHeight and img.shape[1] < maxWidth):
        return img

    # If height smaller than width
    if (img.shape[0] < img.shape[1]):
        scalePercent = maxWidth/img.shape[1]*100
    else:
        scalePercent = maxHeight/img.shape[0]*100

    newWidth = int(img.shape[1] * scalePercent / 100)
    newHeight = int(img.shape[0] * scalePercent / 100)

    newSize = (newWidth, newHeight)
    return cv.resize(img, newSize, interpolation = cv.INTER_AREA)
```

C

Optimize Threshold

Listing C.1: Function to optimize thresholds

```
import csv
import sys
import cv2 as cv
import numpy as np


def threshold_performance(th):


  balance_opt = [0, 0, 0, 0, 0]
  precision_opt = [0, 0, 0, 0, 0]
  recall_opt = [0, 0, 0, 0, 0]
  best_balance = 0
  opt_precision = 0
  db_fileName = TEST_BASE + '/threshold_grande.csv'
  db_file = open(db_fileName, 'r')
  reader = csv.DictReader(db_file)

  #Set the number of minimal matches required
  for min_match in range (1,30):
    db_file.seek(0)
    rela_good = np.array([])
    rela_worse = np.array([])

    # Retrieve data from each data ratio array
    for row in reader:
      try:
        # Calculate min number of matches = threshold * length keypoints
        matches_need = math.ceil(th*int(row['keypoints']))
      except:
          continue


      # If needed, set number of min matches
      if matches_need < min_match:
        matches_need = min_match
```

```
relation = deserialize_descriptors(row['relatie'])
# If label = 1 this is a match so the ratio is set in good,
# else in worse
if row['label'] == '1':
  rela_good = np.append(rela_good, relation[matches_need])
else:
  rela_worse = np.append(rela_worse, relation[matches_need])

# Sort arrays
rela_good = np.sort(rela_good)
rela_worse = np.sort(rela_worse)

max_th = max(rela_good)
min_th = min(rela_worse)


if max_th < min_th:
  print(th, ' is a good threshold')
  opt_th = min_th
else:
  # Loop through all ratio's in rela worse to decide performance
  for dec_th in rela_worse:
    #perfPrint(None)
    FN = 0
    FP = 0
    TN = 0
    TP = 0



    # Calculate confusion matrix
    FN = np.count_nonzero(np.where(rela_good > dec_th))
    FP = np.count_nonzero(np.where(rela_worse < dec_th))

    TP = len(rela_good) - FN
    TN = len(rela_worse) - FP

    # Calculate paramaters
    balanced_acc = 0.5*((TP/(TP+FN))+(TN/(TN+FP)))
    try:
      precision = TP/(TP+FP)
      recall = TP/(TP+FN)
    except:
      precision = 0
      recall = 0

    # Find optimalpoints for precison, balanced accuracy and recall
    if precision > precision_opt[4]:
      precision_opt = [balanced_acc, dec_th, min_match, recall,
      precision, th, TN, TP, FN, FP]
    elif precision == precision_opt[4] and recall >= precision_opt[3]
    and balanced_acc >= precision_opt[0]:
      precision_opt = [balanced_acc, dec_th, min_match, recall,
      precision, th, TN, TP, FN, FP]
```

```
            if balanced_acc > balance_opt[0]:
              balance_opt = [balanced_acc, dec_th, min_match, recall,
              precision, th, TN, TP, FN, FP]
            elif balanced_acc == balance_opt[0] and recall >= balance_opt[3]
            and precision >= balance_opt[4]:
              balance_opt = [balanced_acc, dec_th, min_match, recall,
              precision, th, TN, TP, FN, FP]

            if recall > recall_opt[3]:
              recall_opt = [balanced_acc, dec_th, min_match, recall,
              precision, th, TN, TP, FN, FP]
            elif recall == recall_opt[3] and precision >= recall_opt[4]
            and balanced_acc >= recall_opt[0]:
              recall_opt = [balanced_acc, dec_th, min_match, recall,
              precision, th, TN, TP, FN, FP]


    return precision_opt, balance_opt, recall_opt

balance_optimal = [0, 0, 0, 0, 0]
precision_optimal = [0, 0, 0, 0, 0]
recall_optimal = [0, 0, 0, 0, 0]


for i in range (1,39):
  th = i * 0.001
  precision, balance, recall = threshold_performance(th)


  # Find optimalpoints for precison, balanced accuracy and recall
  if precision[4] > precision_optimal[4]:
    precision_optimal = precision
  elif precision[4] == precision_optimal[4] and precision[3] >=
  precision_optimal[3] and precision[0] >= precision_optimal[0]:
      precision_optimal = precision


  if balance[0] > balance_optimal[0]:
    balance_optimal = balance
  elif balance[0] == balance_optimal[0] and balance[3] >=
  balance_optimal[3] and balance[4] >= balance_optimal[4]:
      balance_optimal = balance


  if recall[3] > recall_optimal[3]:
    recall_optimal = recal
  elif recall[3] == recall_optimal[3] and recall[4] >=
  recall_optimal[4] and recall[0] >= recall_optimal[0]:
    recall_optimal = recall
```

D

Serialization and Deserialization Functions

```
import pickle
import cv2 as cv
import codecs

def serialize_descriptors(descr):
  return codecs.encode(pickle.dumps(descr), "base64").decode()
  # return pickle.dumps(descr, protocol=0) # Protocol=0 is printable ascii
def deserialize_descriptors(ser):
  return pickle.loads(codecs.decode(ser.encode(), "base64"))

def serialize_keypoints(keyps):
  simplified = []

  for keyp in keyps:
    simplified.append((
      keyp.pt,
      keyp.size,
      keyp.angle,
      keyp.response,
      keyp.octave,
      keyp.class_id
    ))

  return pickle.dumps(simplified, protocol=0)

def deserialize_keypoints(simplified):
  keypoints = []

  unpickled = pickle.loads(simplified)

  for simp in unpickled:
    keypoint = cv.KeyPoint(
        x=simp[0][0],
        y=simp[0][1],
        _size=simp[1],
        _angle=simp[2],
        _response=simp[3],
```

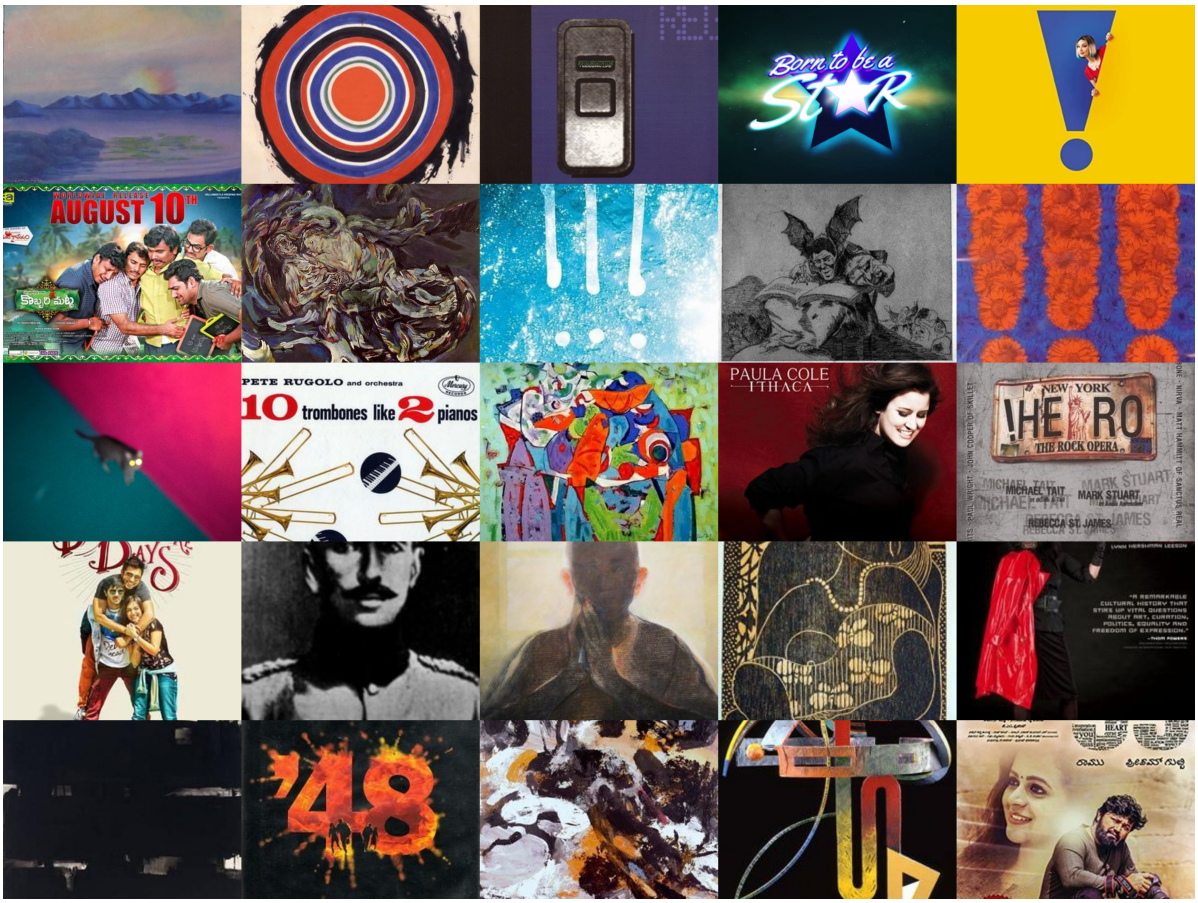
```
            _octave=simp[4],
            _class_id=simp[5])

        keypoints.append(keypoint)

    return keypoints
```

E

Wikipedia Images


Born to be a STAR
AUGUST 10TH
PETE RUGOLO and orchestra
10 trombones like 2 pianos
PAULA COLE
ITHACA
NEW YORK
THE ROCK OPERA
MICHAEL TAIT
MARK STUART
REBECCA ST. JAMES
'48

F

Wikipedia Index and Search Code

Listing F.1: Indexing code

```
import numpy as np
import csv

nextName = read_progress()
sift = cv.SIFT_create()

resetPerf()

def getWikiImageUrls(nameFrom='', batchSize=10):
  wikiApi = "https://en.wikipedia.org/w/api.php"
  response = requests.get(
    wikiApi,params={
        "action":"query",
        "format":"json",
        "list":"allimages",
        "aifrom":nameFrom,
        "ailimit":str(batchSize)
    }).json()

  nextName = response['continue']['aicontinue'] if 'continue' in response else None
  results = {item['name']:item['url'] for item in response['query']['allimages']}
  return results, nextName

BATCH_SIZE = 40

db_fileName = WIKI_DATA + '/data-wiki-new.csv'
print("Creating index at path:", db_fileName)
db_file = open(db_fileName, 'a', newline='')
fieldnames = ['url', 'keypoints', 'descriptors']
writer = csv.DictWriter(db_file, fieldnames=fieldnames)
writer.writeheader()
i = 0
n = 0
while i < 1000: # 5000 batches of 10
# while nextName is not None:
  i += 1
  sourceImageUrlDict, nextName = getWikiImageUrls(nextName, BATCH_SIZE)
  write_progress(nextName)
```

```
  for sourceId in sourceImageUrlDict:
    perfPrint(False)
    sourceImageUrl = sourceImageUrlDict[sourceId]
    sourceImage = getImageFromUrl(sourceImageUrl)
    if sourceImage is None:
      continue
    perfPrint("Image download")
    print("Processing image "
        + sourceImageUrl + " size: "
        + str(sourceImage.shape[0])
        + ' ' + str(sourceImage.shape[1]))
    if sourceImage.shape[0] > 1500 or sourceImage.shape[1] > 1500:
      print("Skipping, too big of an image")
      continue

    keypoints_wiki_data, descriptors_wiki_data = sift.detectAndCompute(sourceImage, N
    perfPrint("needle detect and compute")

    n += 1
    writer.writerow({
      'url': sourceImageUrl,
      'keypoints': serialize_keypoints(keypoints_wiki_data),
      'descriptors': serialize_descriptors(descriptors_wiki_data)
    })
    perfPrint("write csv")


    print("URL: ", sourceImageUrl)
    # print('\tkeypoints: ', serialize_keypoints(keypoints_wiki_data))
    # print(keypoints_wiki_data)
    # print(descriptors_wiki_data)


    #np.savetxt(WIKI_DATA + '/data-wiki.csv', serialized, delimiter=',')

  db_file.flush()
  print('db flush images: %d' % (n))
  perfPrint('db flush')

print("Successfully ndexed ", n, " images")
print("Done, closing db_file")
db_file.close()

print_avergae_runtimes()
```

Listing F.2: IndexingSearch through index code

```
import csv
import sys
import time
sift = cv.SIFT_create()
csv.field_size_limit(sys.maxsize)

resetPerf()

db_fileName = WIKI_DATA + '/data-wiki-new.csv'
```

```
db_file = open(db_fileName, 'r')
reader = csv.DictReader(db_file)

extractTimes = []
totalStart = time.time()

FLAN_INDEX_KDTREE = 0
index_params = dict (algorithm = FLAN_INDEX_KDTREE, trees=5)
search_params = dict (checks=50)

flann = cv.FlannBasedMatcher(index_params, search_params)

needleImageUrls = getImageUrls(ART_NEEDLE_WIKI)
for needleImageUrl in needleImageUrls:
    perfPrint("Starting: " + needleImageUrl)

    current_best = 0
    best_source = getImageUrls(ART_NEEDLE_WIKI)

    needleImage = getImageFromUrl(needleImageUrl)
    perfPrint(False)
    keypointsflann, descriptorsflann = sift.detectAndCompute(needleImage, None)
    perfPrint("Detect and compute needle features")

    plt.imshow(needleImage), plt.show()
    perfPrint("Image show")

    for row in reader:
        print('Comparing to: ', row['url'])

        startTime = time.time()
        # print(row['descriptors'])
        # print(pickle.loads(row['descriptors']))
        try:
            matches = flann.knnMatch (descriptorsflann, deserialize_descriptors(row['desc
            perfPrint("knnMatch")
        except:
            print("ERROR: knnMatch failed")
            continue

        matchesMask = [[0,0] for i in range(len(matches))]

        good_matching = 0

        for i,(m1, m2) in enumerate (matches):
            if m1.distance < 0.5 * m2.distance:
                matchesMask[i] = [1,0]
                good_matching = good_matching + 1

        draw_params = dict (
            matchColor = (0,0,255),
            singlePointColor = (0,255,0),
            matchesMask = matchesMask,
            flags=0 )
        perfPrint("Match filter")
```

```
        print('Number of keypoints:', len(matchesMask))
        print('Number of qualitive matches', good_matching)
        perc = good_matching/len(matchesMask)
        print('Percentage of good matches: ', perc*100, '%')

        if perc > 0 and good_matching >= 1:
            if perc*100 > current_best:
                current_best = perc * 100
                best_source = row['url']
                print('New current best')
        else:
            print('No better match')

        extractTimes.append(time.time() - startTime)

        perfPrint("More filter")
        if perc >= 0.01 and good_matching >= 10:
            print('Found a match!')
            sourceImage = getImageFromUrl(row['url'])
            perfPrint("Image download")
            # flann_matches = cv.drawMatchesKnn(
                needleImage,
                keypointsflann,
                sourceImage,
                deserialize_keypoints(row['keypoints']),
                matches,
                None,
                **draw_params)
            plt.imshow(sourceImage), plt.show()
            # plt.imshow(flann_matches), plt.show()
            current_best = perc * 100
            best_source = row['url']
            keep_going = input('Press x to continue search')
            if keep_going != 'x':
                exit()
        else:
            print('The current best is ', best_source , 'with percentage' , current_b

print("AVG: " + str(sum(extractTimes) / len(extractTimes)))
print(time.time() - totalStart)
```

G

Template Matching: Testcode

Listing G.1: Code to test the feasibility of the template matching technique

```
import sys
import numpy as np
from matplotlib import pyplot as plt
import imutils

# Scales an image down to fit in the bounding maxWidth and maxHeight
# Image should be OpenCV image object
def boundFit(img, maxWidth, maxHeight):
  # If already fitting in side max
  if (img.shape[0] < maxHeight and img.shape[1] < maxWidth):
    return

  # If height smaller than width
  if (img.shape[0] < img.shape[1]):
    scalePercent = maxWidth/img.shape[1]*100
  else:
    scalePercent = maxHeight/img.shape[0]*100

  newWidth = int(img.shape[1] * scalePercent / 100)
  newHeight = int(img.shape[0] * scalePercent / 100)

  newSize = (newWidth, newHeight)
  return cv.resize(img, newSize, interpolation = cv.INTER_AREA)


needleImageUrls = getImageUrls(EIFFEL_NEEDLE_SINGLE)
haystackImageUrls = getImageUrls(EIFFEL_HAYSTACK_SINGLE)

for needleImageUrl in needleImageUrls:
  needleImage = cv.imread(needleImageUrl)
  needleImage_gray = cv.cvtColor(needleImage, cv.COLOR_BGR2GRAY)

  #needleImage = getImageFromUrl(needleImageUrl)
  print("Searching for needle image", needleImageUrl)

  for haystackImageUrl in haystackImageUrls:
    found = None
    print("Searching in image: ", haystackImageUrl)
```

```
haystackImage = cv.imread(haystackImageUrl)
haystackImage_gray = cv.cvtColor(haystackImage, cv.COLOR_BGR2GRAY)
# haystackImage_gray_canned = cv.Canny(haystackImage_gray, 50, 200)

print("Haystack downscale search")
h_needle, w_needle = needleImage_gray.shape[:2]
# Needle has static size in haystack_resize step
for haystack_scale in np.linspace(0.1, 1.0, 40)[::-1]:
  # Downscale haystackImage
  haystackImage_gray_resized = imutils.resize(
    haystackImage_gray,
    width = int(haystackImage_gray.shape[1] * haystack_scale))
  r = haystackImage_gray.shape[1] / float(haystackImage_gray_resized.shape[1])

  haystack_image_preprocessed = haystackImage_gray_resized

  if haystack_image_preprocessed.shape[0] < h_needle
    or haystack_image_preprocessed.shape[1] < w_needle:
    print('Inc size haystack_resize')
    break

  if h_needle < 0.1 * haystack_image_preprocessed.shape[0]
    or w_needle < 0.1*haystack_image_preprocessed.shape[1]:
    print("Needle too small")
    break

  # Apply template matching
  try:
    res = cv.matchTemplate(haystack_image_preprocessed, needleImage_gray, cv.TM_C
  except:
    print('Could not template match: ', sys.exc_info()[0])
    continue

  # Get location if minimum and maximum point of template matched image
  min_val, max_val, min_loc, max_loc = cv.minMaxLoc(res)
  print("Needle width: ",
    needleImage_gray.shape[1],
    " haystack width: ",
    haystack_image_preprocessed.shape[1],
    " Score: ",
    max_val)

  plt.ylim(0, max(
    haystack_image_preprocessed.shape[0],
    haystack_image_preprocessed.shape[0]))
  plt.xlim(0, max(
    needleImage_gray.shape[1],
    haystack_image_preprocessed.shape[1]))
  plt.subplot(121), plt.imshow(needleImage_gray)
  plt.subplot(122), plt.imshow(haystack_image_preprocessed)

  # if we have found a new maximum correlation value, then update
  # the bookkeeping variable
  if found is None or max_val > found[1]:
    print("Found new better match! haystack_resize 1/r:", 1/r, " maxVal: ", max_
```

```
      found = (True, max_val, max_loc, r, res, w_needle, h_needle)

  print("Needle downscale search")
  for needle_scale in np.linspace(0.1, 1.0, 40)[::-1]:
    needleImage_gray_resized = imutils.resize(
      needleImage_gray,
      width = int(needleImage_gray.shape[1] * needle_scale))
    r = needleImage_gray.shape[1] / float(needleImage_gray_resized.shape[1])

    h_needle, w_needle = needleImage_gray_resized.shape[:2]

    # haystackImage_gray_resized_canned = cv.Canny(haystackImage_gray_resized, 50, 200)

    if haystackImage_gray.shape[0] < h_needle or haystackImage_gray.shape[1] < w_needle:
      print('Inc size needle_resize')
      break

    if h_needle < 0.1 * haystackImage_gray.shape[0]
      or w_needle < 0.1*haystackImage_gray.shape[1]:
      print("Needle too small")
      break

    # Apply template matching
    try:
      res = cv.matchTemplate(haystackImage_gray, needleImage_gray_resized, cv.TM_CCOEFF)
    except:
      print('Could not template match: ', sys.exc_info()[0])
      continue

    # Get location if minimum and maximum point of template matched image
    min_val, max_val, min_loc, max_loc = cv.minMaxLoc(res)
    print(
      "Needle width: ",
      needleImage_gray_resized.shape[1],
      " haystack width: ",
      haystackImage_gray.shape[1],
      " Score: ",
      max_val)

    # if we have found a new maximum correlation value, then update
    # the bookkeeping variable
    if found is None or max_val > found[1]:
      print("Found new better match! needle_resize 1/r:", 1/r, " maxVal: ", max_val)
      found = (False, max_val, max_loc, r, res, w_needle, h_needle)

  if found is not None:
    # Extract best match
    (isHaystackResize, _, maxLoc, r, res, w_needle, h_needle) = found
    if isHaystackResize:
      (topLeftX, topLeftY) = (int(maxLoc[0] * r), int(maxLoc[1] * r))
      (botRightX, botRightY) = (int((maxLoc[0] + w_needle) * r), int((maxLoc[1] + h_nee
    else:
      (topLeftX, topLeftY) = (int(maxLoc[0]), int(maxLoc[1]))
      (botRightX, botRightY) = (int((maxLoc[0] + w_needle)), int((maxLoc[1] + h_needle))

    plt.subplot(141),plt.imshow(needleImage_gray)
```

```
    plt.title('Search image'), plt.xticks([]), plt.yticks([]) # Disables axis

    # bottom_right = (max_loc[0] + w_needle, max_loc[1] + h_needle)
    cv.rectangle(
      haystackImage_gray,
      (topLeftX, topLeftY),
      (botRightX, botRightY),
      (0, 0, 255),
      2)
    plt.subplot(142), plt.imshow(res, cmap = 'gray')
    plt.title('Matching Result'), plt.xticks([]), plt.yticks([])
    plt.subplot(143), plt.imshow(haystackImage_gray, cmap = 'gray')
    plt.title('Detected Point'), plt.xticks([]), plt.yticks([])
    # plt.suptitle(_method)
    plt.show()
  else:
    print("found is none")
```

H

Full codes

H.1. ORB-BF

```
import numpy as np
import csv

# Get image URLs
needleImageUrls = getImageUrls(TEST_NEEDLE)
haystackImageUrls = getImageUrls(TEST_HAYST)

# Loop through Needles
for needleImageUrl in needleImageUrls:
  found = None

  needleImage = cv.imread(needleImageUrl)
  needleImage_gray = cv.cvtColor(needleImage, cv.COLOR_BGR2GRAY)


  print("Searching for needle image", needleImageUrl)
  # Loop Through haystack
  for haystackImageUrl in haystackImageUrls:
    print("Searching in image: ", haystackImageUrl)

    haystackImage = cv.imread(haystackImageUrl)
    haystackImage_gray = cv.cvtColor(haystackImage, cv.COLOR_BGR2GRAY)

    orb = cv.ORB_create()


    keypoints_needle, descriptors_needle = orb.detectAndCompute(needleImage, None)

    keypoints_haystack, descriptors_haystack = orb.detectAndCompute(haystackImage, None)


    bf = cv.BFMatcher_create(cv.NORM_HAMMING, crossCheck=True)

    matches = bf.match(descriptors_needle, descriptors_haystack)


    matches = sorted(matches,key=lambda x:x.distance)
```

```
    distant = np.zeros(30)
    for i in range(30):
      single = matches[i]
      distant[i] = single.distance


    # ORB_matches =
    plt.imshow(
        cv.drawMatches(
            needleImage,
            keypoints_needle,
            haystackImage,
            keypoints_haystack,
            matches[:30],
            None,
            flags=2))
    plt.show()
```

H.2. SIFT-BF

```
import numpy as np
import csv
best_source = ''
current_best = 0
sift = cv.SIFT_create()

# db_fileName = BASE + '/Datasets/test_data/data_dist.csv'
# db_file = open(db_fileName, 'a', newline='')
# fieldnames = ['url_needle', 'url_haystack', 'good', 'relatie']
# writer = csv.DictWriter(db_file, fieldnames=fieldnames)
# writer.writeheader()

needleImageUrls = getImageUrls(TEST_NEEDLE)
haystackImageUrls = getImageUrls(TEST_HAYST)


for needleImageUrl in needleImageUrls:
  found = None

  needleImage = cv.imread(needleImageUrl)
  needleImage_gray = cv.cvtColor(needleImage, cv.COLOR_BGR2GRAY)

  keypoints_needle, descriptors_needle = sift.detectAndCompute(needleImage, None)

  #needleImage = getImageFromUrl(needleImageUrl)
  print("Searching for needle image", needleImageUrl)
  #plt.imshow(needleImage), plt.show()

  for haystackImageUrl in haystackImageUrls:
    print("Searching in image: ", haystackImageUrl)

    haystackImage = cv.imread(haystackImageUrl)
    haystackImage_gray = cv.cvtColor(haystackImage, cv.COLOR_BGR2GRAY)

    sift = cv.SIFT_create()
```

```
#perfPrint(False)
keypoints_haystack, descriptors_haystack = sift.detectAndCompute(haystackImage, None)
#perfPrint('SIFT detect and compute')

bf = cv.BFMatcher()

matches = bf.knnMatch(descriptors_needle,descriptors_haystack, k=2)

#Apply ratio test
good = []
for m,n in matches:
    match_i.append(rela)
    if m.distance < 0.6*n.distance:
        good.append([m])


print('Number of keypoints:', len(matches))
print('Number of qualitive matches', len(good))
perc = len(good)/len(matches)
print('Percentage of good matches: ', perc*100, '%')

if perc > 0.06 and len(good) >= 1:

    if perc*100 > current_best:
        current_best = perc * 100
        best_source = haystackImageUrl
        print('New current best')
else:
    print('No better match')

if perc >= 0.01: #and len(good) >= 10:
    print('Found a match!')

    plt.imshow(haystackImage), plt.show()
    #plt.imshow(SIFT_matches), plt.show()
    current_best = perc * 100
    best_source = haystackImageUrl
else:
    print('The current best is' , best_source , 'with percentage' , current_best)

SIFT_matches = cv.drawMatchesKnn(
    needleImage,
    keypoints_needle,
    haystackImage,
    keypoints_haystack,
    good,
    None,
    flags=2)
plt.imshow(SIFT_matches)
plt.show()
```

H.3. SIFT-FLANN

```
import numpy as np
import csv
```

```
current_best = 0


needleImageUrls = getImageUrls(EIFFEL_NEEDLE)
haystackImageUrls = getImageUrls(EIFFEL_HAYSTACK_SMALL)

sift = cv.SIFT_create()


for needleImageUrl in needleImageUrls:
  found = None

  current_best = 0
  best_source = getImageUrls(TEST_NEEDLE)


  needleImage = cv.imread(needleImageUrl)


  #needleImage = getImageFromUrl(needleImageUrl)
  print("Searching for needle image", needleImageUrl)
  plt.imshow(needleImage), plt.show()

  keypointsflann, descriptorsflann = sift.detectAndCompute(needleImage, None)

  for haystackImageUrl in haystackImageUrls:

    print("Searching in image: ", haystackImageUrl)

    haystackImage = cv.imread(haystackImageUrl)


    perfPrint(False)
    keypointsnflan, descriptorsflan = sift.detectAndCompute(haystackImage, None)


    FLAN_INDEX_KDTREE = 0
    index_params = dict (algorithm = FLAN_INDEX_KDTREE, trees=5)
    search_params = dict (checks=50)

    flann = cv.FlannBasedMatcher(index_params, search_params)


    matches = flann.knnMatch (descriptorsflann, descriptorsflan,k=2)


    matchesMask = [[0,0] for i in range(len(matches))]

    good_matching = 0


    for i,(m1, m2) in enumerate (matches):

        if m1.distance < 0.6 * m2.distance:
            matchesMask[i] = [1,0]
            good_matching = good_matching + 1
```

```
draw_params = dict (
    matchColor = (0,0,255),
    singlePointColor = (0,255,0),
    matchesMask = matchesMask,
    flags=0 )

print('Number of keypoints:', len(matchesMask))
print('Number of qualitive matches', good_matching)
perc = good_matching/len(matchesMask)
print('Percentage of good matches: ', perc*100, '%')


if perc > 0 and good_matching >= 1:
    if perc*100 > current_best:
        current_best = perc * 100
        best_source = haystackImageUrl
        print('New current best')
else:
    print('No better match')

if perc >= 0.01 and good_matching >= 1:

    print('Found a match!')
    plt.imshow(haystackImage), plt.show()

    current_best = perc * 100
    best_source = haystackImageUrl
else:
    print('The current best is' , best_source , 'with percentage' , current_best)
flann_matches = cv.drawMatchesKnn(
    needleImage,
    keypointsflann,
    haystackImage,
    keypointsnflan,
    matches,
    None,
    **draw_params)
plt.imshow(flann_matches), plt.show()
```

Bibliography

[1] M. van Geerenstein, P. van Mastrigt, and L. Vergroesen, *Image search engine for digital history: Deep learning approach*.

[2] A. Nanetti, *Engineering Historical Memory*, https://engineeringhistoricalmemory.com, [Accessed 21 May 2021], 2021.

[3] ——, *Engineering Historical Memory*, https://engineeringhistoricalmemory.com/Credits.php, [Accessed 07 June 2021], 2021.

[4] ——, "Defining heritage science: A consilience pathway to treasuring the complexity of inheritable human experiences through historical method, ai, and ml," *Complexity*, vol. 2021, p. 4 703 820, 2021, ISSN: 1076-2787. DOI: 10.1155/2021/4703820.

[5] B. Seguin, "The replica project: Building a visual search engine for art historians," *XRDS*, vol. 24, no. 3, pp. 24–29, 2018. DOI: https://doi.org/10.1145/3186653.

[6] B. Seguin, C. Striolo, I. diLenardo, and F. Kaplan, "Visual link retrieval in a database of paintings," *Springer International Publishing 2016*, vol. 9913, no. 1, pp. 753–767, 2016. DOI: https://doi.org/10.1007/978-3-319-46604-0_52.

[7] F. Condorelli, F. Rinaudo, F. Salvadore, and S. Tagliaventi, "A neural networks approach to detecting lost heritage in historical video," *ISPRS International Journal of Geo-Information*, vol. 9, no. 5, 2020, ISSN: 2220-9964. DOI: 10.3390/ijgi9050297. [Online]. Available: https://www.mdpi.com/2220-9964/9/5/297.

[8] L. Schomaker, *A large-scale field test on word-image classification in large historical document collections using a traditional and two deep-learning methods*, 2019. arXiv: 1904.08421 [cs.CV].

[9] T. van der Zant, L. Schomaker, S. Zinger, and H. van Schie, "Where are the search engines for handwritten documents?" *Interdisciplinary Science Reviews*, vol. 34, no. 2-3, pp. 224–235, 2009. DOI: 10.1179/174327909X441126. eprint: https://doi.org/10.1179/174327909X441126. [Online]. Available: https://doi.org/10.1179/174327909X441126.

[10] T. M. Rath, R. Manmatha, and V. Lavrenko, "A search engine for historical manuscript images," in *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '04, Sheffield, United Kingdom: Association for Computing Machinery, 2004, pp. 369–376, ISBN: 1581138814. DOI: 10.1145/1008992.1009056. [Online]. Available: https://doi.org/10.1145/1008992.1009056.

[11] D. Michaud, T. Urruty, P. Carré, and F. Lecellier, "Adaptive features selection for expert datasets: A cultural heritage application," *Signal Processing: Image Communication*, vol. 67, pp. 161–170, 2018, ISSN: 0923-5965. DOI: https://doi.org/10.1016/j.image.2018.06.011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S092359651830105X.

[12] R. Veltkamp and M. Tanase, "Content-based image retrieval systems: A survey," *Technical report, Utrecht University*, Nov. 2000.

[13] M. Hamilton, S. Fu, M. Lu, J. Bui, D. Bopp, Z. Chen, F. Tran, M. Wang, M. Rogers, L. Zhang, C. Hoder, and W. T. Freeman, *Mosaic: Finding artistic connections across culture with conditional image retrieval*, 2021. arXiv: 2007.07177 [cs.LG].

[14] R. Inbaraj and G. Ravi, "A survey on recent trends in content based image retrieval system," *Journal of Critical Reviews*, vol. 7, no. 11, pp. 961–965, 2020.

[15] M. Yasmin, S. Mohsin, and M. Sharif, "Intelligent image retrieval techniques: A survey," *Journal of Applied Research and Technology*, vol. 12, no. 1, pp. 87–103, 2014, ISSN: 1665-6423. DOI: https://doi.org/10.1016/S1665-6423(14)71609-8.

[16] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part-based models," *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 9, pp. 1627–1645, 2009.

[17] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman, "Object retrieval with large vocabularies and fast spatial matching," in *2007 IEEE conference on computer vision and pattern recognition*, IEEE, 2007, pp. 1–8.

[18] J. Yang, Y.-G. Jiang, A. G. Hauptmann, and C.-W. Ngo, "Evaluating bag-of-visual-words representations in scene classification," ser. MIR '07, Augsburg, Bavaria, Germany: Association for Computing Machinery, 2007, pp. 197–206, ISBN: 9781595937780. DOI: 10.1145/1290082.1290111.

[19] P. G. Max Deutman and O. van Hooff, *An inquiry into the ethics and technology of search engines*, 2021.

[20] Q. Jia, J. Cai, Z. Cao, Y. Wu, X. Zhao, and J. Yu, "Deep learning for object detection and grasping: A survey," in *2018 IEEE International Conference on Information and Automation (ICIA)*, 2018, pp. 427–432. DOI: 10.1109/ICInfA.2018.8812318.

[21] Z.-Q. Zhao, P. Zheng, S.-t. Xu, and X. Wu, "Object detection with deep learning: A review," *IEEE transactions on neural networks and learning systems*, vol. 30, no. 11, pp. 3212–3232, 2019.

[22] J. Ma, X. Jiang, A. Fan, J. Jiang, and J. Yan, "Image matching from handcrafted to deep features: A survey," *International Journal of Computer Vision*, vol. 129, no. 1, pp. 23–79, 2021, ISSN: 1573-1405. DOI: 10.1007/s11263-020-01359-2.

[23] J.-P. Mercier, M. Garon, P. Giguère, and J.-F. Lalonde, *Deep template-based object instance detection*, 2020. arXiv: 1911.11822 [cs.CV].

[24] I. Talmi, R. Mechrez, and L. Zelnik-Manor, *Template matching with deformable diversity similarity*, 2017. arXiv: 1612.02190 [cs.CV].

[25] R. N. Luces, *Template-based versus feature-based template matching*. [Online]. Available: https://medium.datadriveninvestor.com/template-based-versus-feature-based-template-matching-e6e77b2a3b3a.

[26] I. E. Lager, K. Bertels, V. Scholten, E. Bol, C. Richie, and S. Izadkhast, *EE3L11 Bachelor Graduation Project*, 2020-2021. Delft University of Technology, 2020.

[27] M. van Geerenstein, P. van Mastrigt, and L. Vergroesen, *An inquiry into the ethics and technology of artificial intelligence*, 2021.

[28] *About opencv*, Nov. 2020. [Online]. Available: https://opencv.org/about/.

[29] *Opencv template matching*, https://docs.opencv.org/master/d4/dc6/tutorial_py_template_matching.html, [Accessed 19 May 2021].

[30] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *2011 International conference on computer vision*, Ieee, 2011, pp. 2564–2571.

[31] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *European conference on computer vision*, Springer, 2006, pp. 430–443.

[32] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "Brief: Binary robust independent elementary features," in *European conference on computer vision*, Springer, 2010, pp. 778–792.

[33] M. Calonder, V. Lepetit, M. Ozuysal, T. Trzcinski, C. Strecha, and P. Fua, "Brief: Computing a local binary descriptor very fast," *IEEE transactions on pattern analysis and machine intelligence*, vol. 34, no. 7, pp. 1281–1298, 2011.

[34] E. Karami, S. Prasad, and M. Shehata, "Image matching using sift, surf, brief and orb: Performance comparison for distorted images," *arXiv preprint arXiv:1710.02726*, 2017.

[35] T. Deepanshu, *Introduction to brief*, Mar. 2019. [Online]. Available: https://medium.com/data-breach/introduction-to-brief-binary-robust-independent-elementary-features-436f4a31a0e6.

[36] D. Lowe, "Object recognition from local scale-invariant features,[in proc. 7th int. conf," *Computer Vision, Kerkyra, Greece*, pp. 1150–1157, 1999.

[37] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[38] J. S. Beis and D. G. Lowe, "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces," in *Proceedings of IEEE computer society conference on computer vision and pattern recognition*, IEEE, 1997, pp. 1000–1006.

[39] M. R. Kirchner, "Automatic thresholding of sift descriptors," in *2016 IEEE International Conference on Image Processing (ICIP)*, IEEE, 2016, pp. 291–295.

[40] *Introduction to surf (speeded-up robust features)*. [Online]. Available: https://docs.opencv.org/3.4/df/dd2/tutorial_py_surf_intro.html.

[41] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (surf)," *Computer vision and image understanding*, vol. 110, no. 3, pp. 346–359, 2008.
[42] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *Computer Vision – ECCV 2006*, A. Leonardis, H. Bischof, and A. Pinz, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 404–417, ISBN: 978-3-540-33833-8.
[43] T. Lindeberg, "Scale selection properties of generalized scale-space interest point detectors," *Journal of Mathematical Imaging and vision*, vol. 46, no. 2, pp. 177–210, 2013.
[44] N. Ricker, "Wavelet contraction, wavelet expansion, and the control of seismic resolution," *Geophysics*, vol. 18, no. 4, pp. 769–792, 1953.
[45] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, IEEE, vol. 1, 2001, pp. I–I. [Online]. Available: https://ieeexplore-ieee-org.tudelft.idm.oclc.org/document/4270197.
[46] *Brute force algorithms explained*, 2020. [Online]. Available: https://www.freecodecamp.org/news/brute-force-algorithms-explained/1.
[47] D. J. Bernstein, "Understanding brute force," in *Workshop Record of ECRYPT STVL Workshop on Symmetric Key Encryption, eSTREAM report*, Citeseer, vol. 36, 2005, p. 2005.
[48] M. Muja and D. Lowe, "Flann-fast library for approximate nearest neighbors user manual," *Computer Science Department, University of British Columbia, Vancouver, BC, Canada*, vol. 5, 2009.
[49] V. Vijayan and P. Kp, "Flann based matching with sift descriptors for drowsy features extraction," in *2019 Fifth International Conference on Image Information Processing (ICIIP)*, IEEE, 2019, pp. 600–605.
[50] H.-J. Chien, C.-C. Chuang, C.-Y. Chen, and R. Klette, "When to use what feature? sift, surf, orb, or a-kaze features for monocular visual odometry," in *2016 International Conference on Image and Vision Computing New Zealand (IVCNZ)*, IEEE, 2016, pp. 1–6.
[51] P. D. E. Weitz. [Online]. Available: http://weitz.de/sift/.
[52] S. Zivkovic, *Feature matching methods comparison in opencv*, 2021. [Online]. Available: http://datahacker.rs/feature-matching-methods-comparison-in-opencv/.
[53] *Feature matching*. [Online]. Available: https://www.docs.opencv.org/master/dc/dc3/tutorial_py_matcher.html.
[54] *Dataset images and code mentioned in this thesis*. [Online]. Available: https://github.com/Avraamu/EHM-BEP-2021.
[55] *Api reference for the allimages endpoint of wikipedia*. [Online]. Available: https://www.mediawiki.org/wiki/API:Allimages.
[56] GPToday, *F1 monaco grand prix 2021*, [Accessed 06 June 2021], 2021. [Online]. Available: https://www.gptoday.net/nl/fotos/f/60aa701c5a74126deb8974dae9966ec82b4c13aa3f218.jpg.
[57] P. Smith, "Bilinear interpolation of digital images," *Ultramicroscopy*, vol. 6, no. 2, pp. 201–204, 1981, ISSN: 0304-3991. DOI: https://doi.org/10.1016/0304-3991(81)90061-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304399181900619.
[58] R. A. Hummel, B. Kimia, and S. W. Zucker, "Deblurring gaussian blur," *Computer Vision, Graphics, and Image Processing*, vol. 38, no. 1, pp. 66–80, 1987.
[59] A. Bundy and L. Wallen, "Difference of gaussians," in *Catalogue of Artificial Intelligence Tools*, Springer, 1984, pp. 30–30.
[60] S. Bedi, S. Gravelle, and Y. Chen, "Principal curvature alignment technique for machining complex surfaces," 1997.
[61] L. M. Milne-Thomson, *The calculus of finite differences*. American Mathematical Soc., 2000.
[62] Stockvault, *Albert einstein*, [Accessed 06 June 2021, resized to 768x1000], 2021. [Online]. Available: https://www.stockvault.net/photo/200506/albert-einstein#.