Computer Science
COC251
B820004

Feature Extraction for Image Classification

by

James W. J. Barber

Supervisor: Sara Saravi

Department of Computer Science
Loughborough University

June 2022

Abstract

Humans have the innate ability to look at objects and patterns and be able to identify and classify them. Computers however, lack this, they can't just be given an image to say that it is this or that, this is where the area of computer vision comes in. I will be creating a system using two techniques called FAST: Faster and Better, a corner detector and BRIEF: Binary Robust Independent Elementary Features, a descriptor, to extract features in a given image. This system will effectively give a computer eyes and give them the ability to pick out interesting features of images and describe them using binary numbers. You can go on to implement this system in a classification software to allow the computer to classify different objects or scenes in an image.

Table of Contents

List of Figures

List of Tables

1. Introduction

This project is about feature extraction for image classification. This requires using techniques to extract clumps or clusters of pixels that have a resemblance of real-life objects. These features will directly correlate to a set of labels that are use to classify the image. The problems in question have been wildly solved and have a massive variety of techniques already available, I will be looking to combine a set of ideas together to make a full system of feature extraction to classification. My aim is to create a system that able to detect features in an image, and be able to match those with another similar object. This type of extraction can then lead to classification of the features with another system.

To be able to complete this, I will need to research scientific papers and tools that are used for image feature extraction and classification. Some of these techniques will be used together to make a system that will complete the feature extraction. The techniques need to be able to interface with each other so the image can go from a grid of pixels, to having its features extracted, to then be given descriptions.

Most classification systems require the features to be compared to each other so the classifier can increase its accuracy by updating its system as new objects are extracted. This is where the description come in, as the system could get new description and compare them to its database to gather better knowledge about the objects it is classifying.

The project will be based around extracting features and giving them descriptions for another system to be able to classify it, or be able to classify a group of features as objects in a scene. I will be focusing on the extraction of these features and giving them descriptions.

At the end of this project, my aim is to have a working system that takes in an image, and gives a list of features with descriptions. The system will be accurate more than it is fast, but speed will be looked at to see if videos could be used instead of images. My aim is to also increase my knowledge of feature extractions and computer vision as a whole.

2. Literature Review

Distinctive Image Features from Scale-Invarient Keypoints [12]

The paper presents a method called **Scale-Invarient Feature Transform** (SIFT) which is a widely used way of extracting features in images. The features are to be used for reliable matching between different views of an objects or scenes, meaning that no matter the image given, the features will be able to be matched to another image's features (if the image dataset is large enough). Lowe also describes a way that the features could be used for object recognition.

Feature extraction is done is by going through 4 major computational steps to generate a set of features, each more computationally expensive than the last, in which they have to past the first steps before the next, reducing overall computation needed. The steps are as followed: **Scale-space extrema detection, Keypoint localization, Orientation assignment and Keypoint descriptor.**

From the method, the outcome of a 500x500 pixel image gives about 2000 features. These features or SIFT keypoints can be used for matching in different images, and because they are highly distinctive, it gives the correct match for a keypoints to be selected from large databases of other keypoints. These robust keypoints are invariant to different scales, rotation and other distortions, this gives small local features can be matched for small and highly obstructed objects, whilst also large keypoints to be matched under noise and distortions. Having the computation efficient, gives nearly real-time performance for thousands of keypoints in an image.

Light-Weight RefineNet for Real-Time Semantic Segmentation [13]

Here in this paper they adapt an already existing semantic segmentation method called **RefinedNet** to increase the speed, enough for real-time performance. They achieve this, by reducing the existing model by decreasing the number of parameters and floating point operation, causing the algorithm to be less computationally expensive. They call this method **ResNet**.

RefineNet belongs to 'encoder-decoder' segmentation networks, the image is downsized progressively and then upscaled progressively to the original size. But as this approach tends to have a large number of parameters and floating point operations, it is hard to get real time performance, so they focus on the decoder to reduced both, without affecting accuracy too much. To show this, they compare ResNet against widely used semantic segmentation benchmarks.

By reducing those mentioned above, ResNet achieve a performance of 55fps on 512x512 pixel image, whereas the original RefineNet gets 20fps, greatly improving the speed. This method can then be used with classification networks afterwards.

Non-Metric Affinity Propagation for Unsupervised Image Categorization [8]

The method preformed they use in the paper is called affinity propagation (AP), it is used in contrast to a k-centres clustering. Instead of taking a random set of values from the image, called 'exemplars', AP

simultaneously considers all data points as potential exemplars, then repeatedly exchanges messages between those data points to find a good solution.

The exemplars are very useful in the method as they hold a large amount of data, such as dependencies in the image features, as well as being a direct image or image fragment from the original image. This means that with multiple exemplars, you can construct realistic predictions. Thus, able to use them efficiently as pointers into the training data for the categorization.

Using the AP to consider all training cases to produce a good set of exemplars, instead of a fixed set of K exemplars and refining those (this is with the k-centres clustering). It successfully achieves higher classification rates to that of k-centres clustering.

Image Categorization Using Scene-Context Scale Based on Random Forests [9]

The paper looks at image categorization using random forests, they are extremely fast in training and testing, giving near real time performance. The method trains multi-scale texton forests to give clustering and local classification.

Proposed is a new framework that uses scene-context scale, to give better category classification. Scene-context is used on a local group of pixels to give new information, as they very local groups, they give ambiguous features as the colour and texture are insufficient to give pixel classes.

This is then expanded upon by using multiple random forests to estimate scene-context, the paper proposed multi-scale texton forest, which generates textons. Combine this with scene-context scale to improve accuracy of the categorization. With experiments the paper confirmed that the method using multi-scale texton forest and scene-context gives better results for image categorization.

Real Time Face Detection System Using Adaboost and Haar-like Features [16]

What they propose in this paper is using **Haar-like** features to detect faces, upon this they use the **Adaboost** algorithm to improve the classification. The benefit of using Haar-like features is that no matter the input size, the calculation is always constant, as they use integral images.

They start by collecting a database of face and non-face images, then building a bank of weak classifiers from the Haar-like features, boosting those features with Adaboost to create stronger classifiers.

Boosting is shown to be effective for face detection. The system built using Haar-like features and Adaboost boosting, creates an high accuracy real-time face detection software.

Real-Time Document Image Classification Using Deep CNN and Extreme Learning Machines [11]

Here in this paper they propose a way for real-time image classification training and testing for documents. The way they use is to use 2 stages, a deep learning network to extract the features, and then a extreme learning machine (ELM) for the classification. By using two different machine learning algorithms, it can gain real time training and classification.

They describe the problem of having to manually train convolutional neural networks (CNN), which can take hours before feedback. And having a self-learning document analysis system by user feedback would take too long to get accurate results. They propose to use a CNN and an ELM to provide real time training, that allows for automatic feature extraction and training of the classifiers. Using ELM is highly efficient to train with, as it takes a millisecond to train one image, getting real-time results.

Using the ELM classifier with a little pre-training already outperformed the state-of-the-ark systems when the paper was released with only 100 training samples. Training the classifiers with the CNN and ELM

combination only needs 1 millisecond to train per image. This is mostly for the feature extraction, meaning a better model for feature extraction could speed it up.

Bayesian Image Classification using Markov Random Fields [3]

Optimization is important to get higher speeds in image analysis, this paper does that by presenting 3 optimization of simulated annealing, these being; Deterministic Pseudo-Annealing (DPA), Game Strategy Approach (GSA) and Modified Metropolis Dynamics. The first is an unambiguous maximisation problem, that's based on relaxation, which is also used for the standard simulation annealing, but using a different way of finding the maximum. The second uses game theory, using the pixels as the players, and using the Nash equilibrium to get maximised values. The third is a modified version of the Metropolis algorithm, but the global state is random each iteration.

What they get from each algorithm is that each is much faster than simulated annealing, and gives good results for image classification, but the trade off is that they are sub-optimal, meaning they are less accurate. They are all similar in the average case in both speed and accuracy.

Adaptive Deep Learning for Soft Real-Time Image Classification [5]

This paper is proposing a method for getting Convolutional Neural Networks (CNN) for real-time applications. This in itself is hard to do, as having lower amount of layers in the CNN adds much more computation, increasing speed, but also increasing accuracy. But because of the nature of the CNNs, increasing may also decreasing the accuracy. Their solution uses imprecise computation, so if the deadline for the time left to get a result, the accuracy could be lowered but meet the timing required. But as we see above, this could mean increasing the layers, increasing speed, as that also could lower accuracy.

They use the notion of **Pareto optimal**, which means that is the CNN has more layers, it means it also has a higher accuracy to one with less layers. So more time=accuracy for a Pareto optimal CNN. This means that you can rule out non-optimal CNNs for the problem. They also obtain a smaller set of the CNNs, called δ-Pareto optimal, which only have the accuracy defined by a value δ, only those with a higher accuracy are δ-Pareto optimal. The paper derives a way to search and find Pareto optimal CNNs and then use them for an adaptive soft real-time image classification system, that can change to a faster CNN when the deadline will not be meet, at the cost of accuracy.

Real-time Convolutional Neural Networks for Emotion and Gender Classification [1]

Real-time gender and expression detection on human faces is the focus of this paper, they go to explain that service robots in homes and stores for example. depends on interaction between the robot and a human. And because state-of-the-art CNNs require millions of parameters, they take way too long to be used in these types of robots. They propose and implement a general case CNN framework to design a real-time CNN. These CNNs that an implemented are testing to provide real-time results and comparable to a human-level performance in guessing gender and expression.

The architecture has been built entirely to reduce the amount of parameters, therefore making the CNN faster and real-time. The amount is reduce by 80x, but still gives favourable results. The model can also be stacked for multi-class classification, whilst keeping real-time.

Fast SIFT Design for Real-Time Visual Feature Extraction [6]

The papers focus is on **SIFT** which is the process define in **[12]**. In the paper they discuss how the SIFT architecture is not very well usable for a real-time implementation, they propose **Fast SIFT**, which vastly decreases the amount of computation and memory space needed to preform the algorithm, whilst still giving a good number of features per frame in a video.

The problem lies within the Gaussian blur iterations that are require for SIFT, they propose a method that uses a parallel layer of SIFT (LPSIFT) to speed up the computation to real-time.

Achieved was a SIFT based feature extraction (Fast SIFT), that could get 2000 features for 1920x1080 pixel videos at 60fps. It reduces that amount of latency lag that the original SIFT suffered with. The new method also saves features that are partial intermediate results for later use.

Real-time Facial Feature Extraction and Emotion Recognition [7]

To achieve real-time results for facial feature extraction and emotion recognition, they use edge counting and image correlation optical flow techniques, which calculate how the face is moving, then inputting the data into a neural network to classify the emotion. To increase the speed to real time, they use some optimization to the techniques said before, such as edge focusing, global motion cancellation.

They focus on areas of the face that show the most emotion, such as the eyes, eyebrows and lips. They use static feature extraction, then using the change to calculate the motion for the neural network. The whole algorithm is base on another, which is a non-real time, they optimize and change the techniques used to create a real time implementation of it.

Overall, the implementation runs in real-time, with a maximum frame rate of 11.1 frames a second, for videos of a 320 by 240 pixel resolution. With some modifications, the facial detection can be increase to 93% from 70-80%. The average emotion correct classification is 44.8%, with an improve method, it increases to 60%.

SURF: Speeded Up Robust Features [2]

The paper presents a new way to detect and describe features in images. The methods uses integral images for image convolutions. For the detectors of the image, a Hessian matrix is used, which is then used for a distribution-based descriptor. These steps are simplified for the problem they are trying to solve, increasing the speed.

Interest points are selected by the detectors, which are distinctive locations, the best being repetitive, which is when they stay the same under different viewing conditions. It becomes a descriptor if it is robust under image deformations (invariant). These are matched between images. They focus on the features that are invariant to scale and rotation, and sometimes rotation can be moved under certain circumstances.

Using a Hessian-matrix detector and a distribution-based descriptors, they create a fast interest point detection-descriptor scheme, which can be expanded upon for either speed or accuracy.

Real-time Facial Feature Extraction using Statistical Shape Model and Haar-wavelet based Feature Search [17]

The paper proposes a technique for fast facial extraction, that uses a statistical shape model and Haar-wavelets. Using a 2D texture pattern search-and-fitting shceme rather than a 1D system such as Active Shape Model (ASM) it provides more robustness and being faster. The Haar-wavelets allow for images to be lower quality and still be able to find the faces, whilst also being quicker.

The techniques is built for embedded systems, so the requirements are tighter, meaning higher accuracy with lower quality images. They conclude that ASM with its 1D information would not be suitable for the task at hand. So they exploit 2D information for each feature, and using the Haar-wavelets for modelling the local patterns.

In conclusion they found that the employing the two aspects of 2D information and Haar-wavelets, it outperforms ASM, and allows for real-time facial extraction. And with their current hardware, the process took between 30-70ms.

BRIEF: Binary Robust Independent Elementary Features [4]

In the paper they propose a descriptor, named BRIEF, it can be computed using simple intensity difference tests. It can be evaluated by using Hamming distances, which is very efficient. The descriptor is tested against SURF and U-SURF [2].

Descriptors are one of the main features in Computer Vision techniques, so having fast and efficient methods is a must. Using binary strings you can very effectively use to find similarities, but to get them from original methods such as SIFT [12] or SURF, you need to do further processing to make the binary strings. Whereas BREIF computes the image directly into these binary stings, they are obtained using intensity differences. Because of this, BREIF outperforms SURF and other state-of-the-art descriptors, in both speed and recognition rates. The only downside is that BREIF descriptions are not invariant to large in-plane rotations.

All this allows BREIF to be very effective for real-time matching, even in systems with limited capabilities.

ORB: An Efficient Alternative to SIFT or SURF [15]

To achieve a full real-time detector and descriptor system, this paper proposes to use the FAST keypoint detector and BREIF descriptors, name ORB (Orietned FAST and Rotated BRIEF). As the initial BREIF algorithm isn't rotation invariant, this paper proposes a change to make it so, and making a learning method for de-correlating BREIF features, to give better performance in nearest-neighbour applications. As well as a fast and accurate orientation component to FAST.

From this, they define a new descriptor, ORB, and showed that it outperforms other popular descriptors. Including being 2x as fast as the popular SIFT, and also less affected by noise, a similar matching performance and all in real-time situations.

Faster and Better: A Machine Learning Approach to Corner Detection [14]

Here in this paper, they propose a new way to detect corner in an image, one that utilizing machine learning to be able to run at very high speeds, so much so that it can run faster than real-time speeds. As detection is in many areas of computer vision the first thing you have to do, having a quick and accurate feature detector is very much sought after. Because many areas are shifting mediums to videos, the demand for a real-time speed detector was at a high, as many of the popular existing detectors run at below frame rate speeds.

Fast uses machine learning to accelerate the process of finding these features, and by using heuristic tests to find the corners, they achieved in creating a algorithm for detecting corners at an unmatched speed at the time.

3. Methodology

3.1 BREIF, Binary Robust Independent Elementary Features

BRIEF or Binary Robust Independent Elementary Features, is a commonly used detector and descriptor, used for its speed and high regonition rates. It is often considered against SIFT [12] and SURF [2], for the purpose of matching. BREIF has an edge over SIFT as its relative simplicity gives it fast computation at the cost of invariance of rotation. SURF is a better comparison, as that already addresses the speed issue of SIFT, but memory becomes an issue as the descriptor's storage is large when in the millions. [4]. Here I talk about the methods of the standard BREIF, and also a variation that addresses the issue of rotational invariance called, Rotated BRIEF.

3.1.1 BREIF

For this method, I will be using the results of the paper [4], where the method was first introduced.

Inensity Calculations

BREIF uses simple intensity comparisons in image patches, to create features and to be classified. These can then go on to be used in classification methods, which will be shown later in the report. These intensity comparison can by shown by the following test τ on patch p of size $S \times S$:

$$\tau(p; x, y) = \begin{cases} 1, & \text{if } p(x) < p(y) \\ 0, & \text{otherwise} \end{cases}$$

where $p(x)$ is the pixel intensity. By choosing a set of $n_d(x, y)$-locations, this defines a set a binary tests, which can be shown as the n_d-dimensional bit-string:

$$f_{n_d}(p) := \sum_{1 \leq i \leq n_d} 2^{i-1} \tau(p; x, y)$$

where $n_d = 128$, 256 and 512, these values give a good compromises between speed, storage and recognition rates.

Image Smoothing

Before using the equation above on the pixel, the image needs to be smoothed. This is because we are working with single pixels in an image, this means the values will be *highly* effected by noise in the image, smoothing the image greatly reduces the amount of noise in the image. For the smoothing, we go with Gaussian Smoothing, or Gaussian Blur. For this, we convolve the image with the Gaussian function:

$$G = (x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}$$

where σ is the varience, the higher this number, the more blur will be done on the image. For more difficult matching, the higher the value should be. In the paper they test different values of σ, and found that a value of 2 is the optimum value. The kernel of the convolution is also decided to be 9 x 9 to achieve the best results.

The convolution is this method is constant throughout any image, meaning a Gaussian function matrix of 9 x 9 can be formed, which can be used on each 9 x 9 kernal on the image to create the blur. This will save computation time as the Gaussian function will only have to run 9^2 or 81 times for each value in the Gaussian function matrix. This creates this matrix:

$$\begin{bmatrix} 0.0007 & 0.0017 & 0.0033 & 0.0048 & 0.0054 & 0.0048 & 0.0033 & 0.0017 & 0.0007 \\ 0.0017 & 0.0042 & 0.0078 & 0.0114 & 0.0129 & 0.0114 & 0.0078 & 0.0042 & 0.0017 \\ 0.0033 & 0.0078 & 0.0146 & 0.0213 & 0.0241 & 0.0213 & 0.0146 & 0.0078 & 0.0033 \\ 0.0048 & 0.0114 & 0.0213 & 0.0310 & 0.0351 & 0.0310 & 0.0213 & 0.0114 & 0.0048 \\ 0.0054 & 0.0129 & 0.0241 & 0.0351 & 0.0398 & 0.0351 & 0.0241 & 0.0129 & 0.0054 \\ 0.0048 & 0.0114 & 0.0213 & 0.0310 & 0.0351 & 0.0310 & 0.0213 & 0.0114 & 0.0048 \\ 0.0033 & 0.0078 & 0.0146 & 0.0213 & 0.0241 & 0.0213 & 0.0146 & 0.0078 & 0.0033 \\ 0.0017 & 0.0042 & 0.0078 & 0.0114 & 0.0129 & 0.0114 & 0.0078 & 0.0042 & 0.0017 \\ 0.0007 & 0.0017 & 0.0033 & 0.0048 & 0.0054 & 0.0048 & 0.0033 & 0.0017 & 0.0007 \end{bmatrix}$$

This matrix will be multiplied and summed up with each pixel in the kernel of the image, those pixels outside of the image will be assumed to have a value of 0. This creates a new pixel value that has the values of its neighbours also affecting its weight, this gives the smoothing/blur effect.

Pixel comparison

When choosing the pixels (x_i, y_i) to compare with each other, you can either go for a pattern, or just choose randomly. The BREIF method uses the isotropic Gaussian distributon, Gaussian(0, $\frac{1}{25}S^2$) where S is the length and width of the image patches. The paper also found that to get the best recognition rates, $\sigma^2 = \frac{1}{25}S^2$.

Hamming Distance for matching

Hamming distance is the number of differences in two bit-strings. The more digits that are different, the higher than distance. For example: **0110101** and **1010101** have a Hamming distance of **2**. This is because only the first two digits of the bit-string are different.

As we are creating a bit-string from the intensity comparisons, we can effectively use the Hamming distance to easily and extremely quickly match two bit-string together. This allows the method to quickly determine if two images are similar, which wil be the basis of the image classification techniques later on in the paper.

3.1.2 Rotated BRIEF

For still images, rotational invariance is not a problem, as the images don't rotate freely. But as the project is on feature extraction for image classification, this can rely on objects being classified no matter their orientation, so having rotational invariance is necessary to get the most features in the image. Rotated BRIEF was created to address that issue, propose in [15].

To make BREIF invariant to plane roation, it is efficient to steer it in relation to the orientation of the keypoints. To do this, you define any set of features of n binary test, each at location (x_i, y_i), the matrix being size $2 \times n$:

$$S = \begin{pmatrix} x_1 & \dots & x_n \\ y_y & \dots & y_n \end{pmatrix}$$

To ajust this matrix, you use the patch orientation θ and do matrix multiplication on the matrix above with the corresponding rotation matrix R_θ, this makes the steered version of S, S_θ:

$$S_\theta = R_\theta S$$

With this, the BREIF string contructor shown in **3.2.1** as $f_{n_d}(p)$ becomes:

$$g_n(p, \theta) = f_n(p) | (x_i, y_i) \in S_\theta$$

The function basically says that the test only includes the x and y that are in the new steered version of S. The angels used are reduced to increments of $2\pi/30$, or 12 degrees, with this you can construct a lookup table of precomputed BREIF patterns.

3.2 Keypoint Detectors

Before we use the BRIEF descriptor shown above, we need to identify the keypoints of the image. To do this, we need to use what is called a detector which scans the image and picks out the keypoints in the image, at first there will be alot of points that will be useless, but there are some techniques you can employ to reduce the amount to those which are useful.

3.2.1 FAST, Features from Accelerated Segment Test

The FAST detector is a corner detection algorithm, it works by having a circle of 16 pixels around the current pixel (see figure 3.1), called the corner candidate, x and classifies it as a corner, or not a corner. The way it does this is by checking the intensity of the pixels in the circle, checking the inner sections first than protruding outwards. The maths from [14] is as follows;

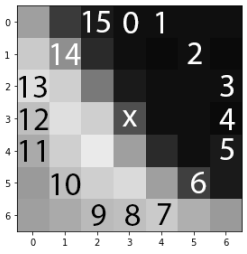


Figure 3.1: Pixel Comparison

You can classify a corner candidate x, as a corner if there exists a set of n contiguous pixels in the circle that are brighter than the intensity of the candidate I_x, plus a threshold value t, or all darker than $I_x - t$. The value of n can change, this value decides how many pixels need to be contiguous, the higher, the harder it is to be classified as a corner. The value chosen in [14] was 12, as this is a nice balance between speed and correctly identifies most non-corners. Whereas 12 was chosen in the beginning, upon testing they found that a value of 9 gave better performance when running the algorithm. I will be using a set of 9 continuous pixels in my project, this is also called FAST-9.

A technique called the High-Speed Test uses the fact that the pixels have to be continuous, meaning rules can be used to remove some of the possibility before having to check the full circle. Each pixel in the circle of x, is examined in stages, with pixels 1 and 9 being examined, if both of these are within I_x +/- t then x cannot be a corner, as it would be impossible to have a continuous n set of pixels. Similarly if 1 or 9 are not within t, then pixels 5 and 13 are considered. If at least 3 of these pixels, 1, 3, 5 and 13, are either brighter or darker than I_x +/- t, respectively, then you can examine all the pixels in the circle for a contiguous set. If there is not at least 3, then it cannot be a corner. Where n

The algorithm is fairly simple, just testing the circle around for continous n points (FAST-n), wheras this is a very effective and fast way to detect corners, it also doesn't address some weaknesses. Two of the weaknesses that are described I will not be using the solotution in this project, these being weaknesses 1 and 2. I will be using the weakness 3, "Multiple features are detected next to each other", this means that there are redundant features being detected.

Nonmaximal Suppression

Having features next to each other isn't necessarily a bad thing, but it will increase the amount of computation for subsequent steps as not get much is the way of results. For this non-maximal suppression is used to reduce the amount of features that are detected. For this, each corner needs to gain a strength, for FAST, this can be done by having the strength equal to the maximal value of the threshold for which that point can be detected as a corner.

For example, the corner strength of a point that has intensity 130. In the 9 pixels (which are darker) that are used to classify as it as a corner, the maximum value was 100. This would mean that the threshold could be up to 30, and still be detected as a corner by FAST-9. This would be the corner strength. It would work similarly if the 9 pixels were brighter, but instead using the minimum value.

Using this, we can create a matrix, the same size of the image, where if (x, y) in the image is a corner, (x, y) in the new matrix is 1, and 0 if not a corner. This is called the corner matrix. Using this and the corner strength, you can find the point where t changed the corner value from 1 to 0, this is achieved via bisection, and as t is discrete, it is a binary search algorithm.

The other method shown in the paper I will not be using.

3.2.2 Oriented FAST

Similar to BRIEF, FAST also has no rotational invariance in the standard algorithm, the paper [15] suggested a way to give FAST a rotational invarience, this is important as objects in different images could be at a different orientation. As defined in the paper, it uses intensity centroid calculations with the moments of the patches to define the orientation of the patch, with the moment of the patch defined as:

$$m_{pq} = \sum_{x,y} x^p y^q I(x, y)$$

where the values of x and y are those in the brief pattern and $I(x, y)$ is the intensity of the point. With the moments we can calculate the centroid of the patch:

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right)$$

With this, you can construct a vector from the corners center, O to the centroid, OC. With which you can obtain the orientation of the patch:

$$\theta = \text{atan2}(m_{01}, m_{10})$$

This then gets used for the patch orientation in the brief descriptors.

4. Desgin

The design on the program dictates what I will be aiming for when implementing the algorithms I have learnt. In my project I will be designing a feature extraction system following ORB, Orietned FAST and Rotated BRIEF, these are the techniques that I have described in the methodology chapter. There are 4 main parts to the system that I am going to implement to create ORB, these are;

- **Orietned FAST Corner Detector**
- **Rotated BRIEF Feature Discriptor**
- **Matching System for the BREIF features**
- **Plotting Functions to Visualise the Process**

Each of these sub systems for will create the process from image to a set of features, and be able to match these features to another image's features. With the visualisation giving us a way to view the results easily.

4.1 FAST Cornor Detectors

The FAST detector will be a set of functions that aim to generate points of interest for the next part of the system (BREIF descriptors), shown in the methodology this is by looking at the surrounding 16 pixels in a circle around the canidate (see figure 3.1). So the design on the functions are not complex, they need to be able to quickly and efficiently extract the corners from a given image. Then use nonmaximal suppression to remove features that are near each other to reduce the amount of duplicates.

The system will include 3 sections, **the corner detector**, **giving the feature a strength** for the nonmaximal suppression and the **nonmaximal suppression**. These 3 will achieve in extracting corner features from an image.

4.1.1 Corner Detector

The main part of the corner detector will be going through the 16 pixels around the candidate pixel to see if there is 9 continuous pixels. This is quite an extensive task so I will use the high speed test that was defined in the methodology, where you check 2 pixels, 8 positions apart to see if it can be considered.

So the corner detector will iteratively go through the image, picking out canidate pixels that satisfy the 9 continuous darker or lighter by the threshold. Those that satisfy will get put into a list of all the successful candidates, now being named keypoints. This can be shown in the pseudo code below:

```
1: t ← Threshold Value
2: for all Pixels in Image do
3:     found ← False
4:     if High Speed Test Success then
5:         for Pixel in Surrounding Circle do
6:             if Current Pixel is Darker by t then
7:                 Check next 8 Pixels are Darker by t
8:                 found ← True
9:             else if Current Pixel is Lighter by t then
10:                Check next 8 pixels are Lighter by t
11:                found ← True
12:            end if
13:        end for
14:        if found then
15:            Add Candidate Pixel to Keypoints
16:        end if
17:    end if
18: end for
```

This pseudo code will form the basis of the corner detector. Its relatively simple, just checking the surrounding pixels to make sure there is 9 continuous darker or lighter pixels, and to not consider the candidate as a corner otherwise.

To make the corner detector invariant to rotation, we need to add intensity centroid calculations on the corner.

4.1.2 Corner Strength

For the nonmaximal suppression, you have to find the value of the maximum value of the threshold, t, where the keypoint, p, can still be considered a corner. To do this, depending on the whether the corner was lighter or darker by the threshold, you can compute the maximum value of the threshold to be:

$$\text{cornerStrength} = \begin{cases} \min(\mathit{pixels}) - I(p), & \text{if } p \leftarrow \text{darker} \\ I(p) - \max(\mathit{pixels}), & \text{if } p \leftarrow \text{lighter} \end{cases}$$

where pixels are the continuous 9 pixel intensities that were found to make p a corner, $I(p)$ is the intensity of the keypoint pixel and $p \leftarrow$ darker/lighter is whether the pixels consisted of 9 darker or lighter pixels.

This function can be placed into the algorithm for the corner detector shown above, to not have to run through all the keypoints again and have to pass the information through to another function, it can be placed in between lines 15 and 16 so it only runs when a corner is found.

4.1.3 Nonmaximal Suppression

Nonmaximal suppression is used to trim the amount of keypoints as most keypoints will be next to each other, detecting the same features. This is where we can use the corner strength we defined to remove some of the weaker features, the method I will be using is different from that which is described in the methodology as I found an easier way to implement nonmaximal suppression. This method I found while searching for already implemented version of FAST-9 [10]

This method uses a corner matrix with the corner strength instead of 1 or 0. This means the corner matrix, we first initialize a zero array, with the same size of the image. In this array you place the corner strength of each corner detected, where (x, y) in the corner matrix is the cornerStrength. You then check the neighbours of the current corner in the corner matrix, if there is a corner with a higher corner strength,

then the current corner is no longer considered as a keypoint. This is effectively the same as the method described, but without having to change t by different amounts and gives a similar result.

You can show the algorithm in pusedo code below:

```
for all Corner in Corner Matrix do
    N ← Neighbourhood of Corner
    for all neighbour in N do
        if corner < neighbour then
            corner = 0
        end if
    end for
end for
```

With this, all the remaining corner in the matrix are the keypoints in the image. These keypoints should all contain different corners and features of the image.

4.2 Rotated BRIEF Feature Descriptor

Brief Features are computed by having tests on the intensity, having a bunch of these tests you can create a bit string that can be used for matching. The patches that the pairs come from are centred around the keypoints that come out of the FAST-9 system. The image that the BREIF features work on has to be blurred, this is due to binary descriptors being very influenced by noise. The BRIEF system will consist of **blurring the image**, **creating the brief feature descriptions** and **lookup table of BRIEF patterns**.

4.2.1 Blurring the Image

As shown in the methodology, I will be using the Gaussian Blur on the image. The algorithm for this is well documented and most image analysis libraries contain a guassian blur/filter, I will attempt to create my own. To do this, I will run the filter matrix that is in the methodology over the image, with each pixel becoming the sum of the product of the neighbouring pixels and the Gaussian function. In the methodology its described using the sigma value 2 to give the best results, so that is what I will be using.

4.2.2 Creating the BRIEF Descriptors

The BRIEF features come form tests preformed on n_d pairs, fo create a n_d length bit-string, where n_d is the number of bits in the bit-string. The value could either be 128, 256 or 512. I will be using $n_d = 256$, as that is what is used in the ORB system. These test pairs are generated from the Gaussian distribution, with sigma being calculated from $\sigma^2 = \frac{1}{25}S^2$, which gives the best recognition rates. S is the size of the patch the pairs come from, the ORB system in [15] uses a patch size of 31 x 31, meaning $S = 31$, this also gives $\sigma = 6.2$.

The pairs in the test patch could have values that would be either outside the patch, or outside the image. In the case that a pair is outside of the patch, its is bounded to the edge of the patch, so the maximum value of the Gaussian distribution to be ± 15 as the patch is 31 x 31. Similarly if the pair was to be outside the image, it is bounded to the edge of the image.

The bit string is constructed using the test, $\tau(p; x, y)$ in the methodology. Each patch uses these pairs and the test to create the binary descriptor for that feature, every keypoint has one of these bit-strings to describe the feature. The algorithm can be shown as:

```
for all k ← Keypoints do
    bitString ← None
    for all test pairs p(x) and p(y) do
```

```
        if p(x) < p(y) then
            bitString ← bitString + 1
        else
            bitString ← bitString + 0
        end if
    end for
    add bitString to descriptor for k
end for
```

The algorithm allows for rapid descriptions of all the features in the image, these descriptors will be used for matching in this project.

4.2.3 Lookup table of Brief Features

The rotated BRIEF comes from the BRIEF patterns that are pre-computed in the form of a lookup table, with each pattern being rotated by $\frac{2\pi}{30}$ or 12 degrees. And the intensity centroid patch orientation will say which pattern to use. As it would be expensive to always create a new pattern, the orientation is discretized to those that are in the lookup table.

When getting the Gaussian distributed test pairs, I will calculate the intensity centroid and find the patch orientation, then find which pattern to use.

4.3 Matching System

A way to compare image similarities, as well as try to match features that turn up in different images. As described in the methodology, this is using hamming distance as we are using binary strings. Which allows for very quick and efficient matching. There are a few ways you could go about choosing how to match each feature with another. The first way would be to just use a greedy nearest neighbour search, where each feature in the first image tries to match with the lowest distance between them, then repeat this for each feature in the first image. This is the simplest method as it doesn't require the features to have any other knowledge of what features have matched previously.

To make sure the features have a low distance to get good recognition rates, a limit can be introduced to reduce the ammount of matches that have low recognition rates, which is indicated by high distance. Different limits will be tested to see which gives the best matches.

The algorithm can be shown as:

```
L ← Limit
for all x ← features in first image do
    match ← False
    currentDistance ← L + 1
    for all y ← features in the second image do
        d ← HammingDistance( x, y)
        if d < L then
            match ← True
            if d < currentDistance then
                currentDistance ← d
                currentMatch ← (x, y)
            end if
        end if
    end for
    if match then
        add currentMatch to matches
```

17: **end if**
18: **end for**

Using *currentDistance* it allows for only smaller distances to be the new match to the current x feature, but that will always be under the limit that is set. This algorithm should then give a list of matches with features that share similarities.

4.4 Visualisation

The visualisation is purely for user interaction and inspection of the process, I will be using a plotting library that allows me to show/save the images that the program is running on. The visualisation system needs to be able to both show image features that have been found, and also show the matches between those images.

The matching image visualisation will have the images side by side, with all the features and lines connecting the matched features, this will easily allow a user to view the algorithms output.

5. Implementation

The implementation will show me going through the process of actually creating the code for the system, this will include my decisions I made along the way regarding the speed and accuracy of the code, and how the code/functions are laid out.

For the implementation I will be using **Python**. I have chosen this becasue it is a language I know well and gives good performance overall. I could've used C, as it would've given a better performance, but it loses a lot of the quality of life that python gives. For the IDE I will be using, I have gone with spyder that is launched from the anaconda program, as it gives me an easy way to install the libraries I need. And a lot of the libraries are pre installed as spyder is a scientific IDE.

5.1 Libraries

Throughout the project I will be using some libraries that will help and speed up both the time the program runs, as more efficent code that what I will be able to write, and also the time needed for me to program the system. I will not be using libraries for the algorithm I have described in the design as I am attempting to implement these, I will however be using it for handling the images and some of the basic functions, such as hamming distance.

5.1.1 Numpy

Numpy is a general use mathematics library that gives more efficent and more complex functions not included in standard python. One of the main uses I will be using is numpy.arrays, as python doesn't have arrays only lists, better preformance and storage is achieved using the arrays. This will be used for storing the images as numpy.uint8 arrays, the type being uint8, gives all the values a range of 0 to 2^8 which is 0-255, the same as the colours for images.

5.1.2 Skimage

Skimage is an image processing library that has many functions to process images in python. One of the main reasons to do this is to load and save images to and from the program, this is using the skimage.io package , which have imread, imshow and imsave.

It also has the functions to adjust the images into grey and colour scales, using the skimage/color package, this changes the values into float values between 0 and 1. I will be using the skimage.exposre package to rescale the floats to 0-255 scale when working with the images.

The library also comes with the draw package, which will allow me to easily draw shapes onto the images for the visualisation part of the program.

5.1.3 Gmpy2

Gmpy2 is a library that has a specific use, as python instead doesn't allow the use of proper bit strings (the closest you could do would be a list or string, both of which uses more stroage), gymp2 has the xmpz package, or multiple-precision integers, this is the advanced version which allows for each bit to be changed, rather than being immutable. This means I can create and change bit strings in python and use the hamdist function within gmpy2 for extremely fast distance calculations.

5.2 General Functions, Assumptions and Importing Libraries

There are a few things that I first need to define so explaining can be easier. First I import all the libraries that will be used in the program, these being descirbed above.

Figure 5.1: Importing Libraries

```
from skimage import io, filters, color, draw, exposure
from gmpy2 import xmpz, hamdist #allows efficent bitstring manipulation
import numpy
import time
```

I will be using the various packages of skimage for the image handling of the program. This includes the reading from files, see figure 5.2, converting the images to grey scale and transforming the values to different ranges. The range I will be using will be 0-255, which are the values for the Numpy.uint8 class also.

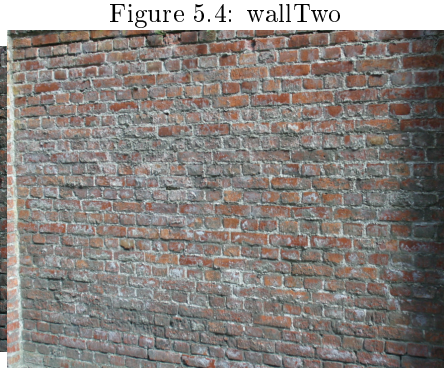
Figure 5.2: Reading Images into the System

```
wallOne = io.imread("wall/img1.ppm")
wallTwo = io.imread("wall/img2.ppm")
```

To read the images into the program, I use the skimage.io package, which allows input/output to and from files and the program, I use io.read to read the two images wallOne and wallTwo. The io.read reads the image as a Numpy.array array of size (imageHeight, imageWidth, channels) where channel is either 1 or 3, depending if the image is greyscale or RGB. In my case, the images wallOne and wallTwo have sizes of (700, 1000, 3) and (680, 880, 3), respectively. The array has type Numpy.uint8.

Figure 5.3: wallOne

Figure 5.4: wallTwo



These two images, shown in **figure 5.3** and **figure 5.4**, are used in the ORB paper [15], and will be used throughout the project as testing images to see check if the algorithms seem to be working the correct way.They are pictures of the same wall, where wallOne is near straight on, and wallTwo is slightly rotated to one side, meaning they are different images of the same thing, meaning matching is appropriate.

Figure 5.5: Convert RGB Image to Greyscale

```
def imageToGrey(img):
    img = color.rgb2gray(img)
    img = exposure.rescale_intensity(img, out_range=(0,255))
    img = numpy.int64(numpy.uint8(img))
    return img
```

There is a general assumption of having the image be greyscale when working with it, this is because it is easier to get intensities with only 1 channel in the image, for this I create the function **imageToGrey**. Using skimage.color I can turn the image into greyscale, but this gives out a scale of 0-1 and a type of Numpy.float64, where I want a scale and 0-255 and Numpy.int64, this is where skimage.exposure comes in and I convert the image into 0-255 and change the type with Numpy.unit8, to give whole values, then change to Numpy.int64 so its easier to work with. This is used anytime the image must be greyscale before the algorithm has to work, this includes the FAST detector and the BRIEF descriptors, as being able to run the functions separately will be useful for testing.

5.3 FAST Corner Detection

5.3.1 Corner Detector

The first step of the system is to locate the corners of the image, so we can get the keypoints to feed into the Brief system. As outlined in the design there is 3 systems you need to implement to achieve a working system. I start by creating the corner detector, the main bulk of the system is a for loop going through the all the pixels in the image whilst checking them with the high speed test to see if they are eligible to be a candidate.

Figure 5.6: Start of the Corner Detector

```
def findNContPixels(image, keypoints, point, N = 9, threshold=30):
    '''corner detection algorithm, FAST-n, uses a 16 pixel circle around the current pixel,
    and then tries to find a n continuos of either strictly darker or lighter pixels, which
    classifies as a corner'''

    pointInt = image[point[0], point[1]]
    pointMax = pointInt + threshold
    pointMin = pointInt - threshold
    points = circle16Pixel(image, point)

    contNine = numpy.array(range(N))

    for i in range(7):
        if abs(points[i] - pointInt) <= threshold and abs(points[i+8] - pointInt) <= threshold:
                #print("7, 15")
            return keypoints
```

First I create a function that takes in the image and a pixel location, and runs the high speed test to check the eligibility. I preform some basic calculations to find the intensity, and then the upper and lower limits of the surrounding pixels to be considered darker or lighter than the candidate pixel. As well as retrieving the points around the pixels with the function **circle16Pixel**. When testing which points made the high speed test faster, I found that if I iteratively checked all pairs of pixels that were 8 apart, instead of those defined in the high speed test, I got a faster performance.

The inputs to the functions are the image, a pass through for the keypoints so more can be added and which point is the candidate. I have added N, which is 9 in my case, but in case it was to change I added it as a variable, as well as the threshold value to which the upper and lower limits of being classified as darker or lighter occur.

Figure 5.7: Find the Pixel surrounding the Candidate

```
def circle16Pixel(image, point):
    points = [
        image[point[0] - 3, point[1]], #### 1
        image[point[0] - 3, point[1] + 1],
        image[point[0] - 2, point[1] + 2], ######
        image[point[0] - 1, point[1] + 3],
        image[point[0], point[1] + 3], ## 5
        image[point[0] + 1, point[1] + 3],
        image[point[0] + 2, point[1] + 2], ###
        image[point[0] + 3, point[1] + 1],
        image[point[0] + 3, point[1]], #### 9
        image[point[0] + 3, point[1] - 1],
        image[point[0] + 2, point[1] - 2], ######
        image[point[0] + 1, point[1] - 3],
        image[point[0], point[1] - 3], ## 13
        image[point[0] - 1, point[1] - 3],
        image[point[0] - 2, point[1] - 2], ###
        image[point[0] - 3, point[1] + 1]
        ]
    return points
```

The **circle16Pixel** receives the image and the point of which I want to retrieve the surrounding 16 pixels, and then returns them as a list. The order of the pixels in the list is the same that was in the methodology, see figure 3.1.

Figure 5.8: Classifying if Candidates are darker or Lighter

```
#find nine contigous points
for i in range(len(points)):
    #print()
    intensity = 0 # 1 == darker, 2 == brighter, 0 = Similar
    corner = True
    #print(points[i])
    if points[i] >= pointMax:
        intensity = 1
    elif points[i] <= pointMin:
        intensity = 2
    else:
        intensity = 0
        #print("next")
        continue
```

To continue the algorithm we iteratively go through each point in the surrounding 16 pixels, we have to classify if the current start pixel in the circle is darker, lighter or similar to the canidate. I do this in **figure 5.8** by first creating a for loop for the points in the circle, and checking whether the pixel is darker or lighter by the limits then setting the intensity as such. If the pixel is within the threshold, then the current pixel is no longer used as the start and the loop continues to the next.

I also set a boolean variable *corner*, which is by default set to $True$ and will get changed later if the corner if there is not another 8 pixels whiuch are also either darker or lighter.

Figure 5.9: Checking for 9 Continuous Pixels

```
contNine[0] = points[i]
for j in range(1, N): #check next 8 pixels
    #print("comp:", points[((i + j) % len(points))])
    contNine[j] = points[((i + j) % len(points))]
    if intensity == 1: #if lighter
        if points[((i + j) % len(points))] < pointMax:
            #print("yes")
            corner = False #is not corner if not 9 contigious
            break
    elif intensity == 2: # if darker
        if points[((i + j) % len(points))] > pointMin:
            #print("no")
            corner = False
            break
```

In **figure 5.9**, I show the loop that finds if there is 9 (or N) pixels in a row to classify if the candidate pixel is a corner. I set up a list that will contain the N continous pixels, and each pixels that is checked is added to the list.

Depending on if the starting pixel was darker or lighter, the program will continue to check if the next $N-1$ pixels are also darker or lighter. Due to the surrounding pixels being in a list, if the program was to see if the continuous N pixels starts at the 14th position, it would have to check if the pixel in the 1st position in the list is same intensity, so I used the modulo of the position to wrap it back around.

The 2 if statements are checking if the point is **not** either darker or lighter, if they are found to be not, then the loop breaks and the next starting pixel is used. The *corner* boolean is also set to $False$, this is used in **figure 5.10** where you see when the loop either ends and there is N continous lighte of darker pixels, or the current starting position leads to less than that and the loop is broken. If the loop just ended, *corner* is set to $True$ still, and the code in 5.10 can run.

Figure 5.10: Adding corner as a Keypoint

```
    if corner:
        if intensity == 1:
            cornerStrength = min(contNine) - pointMax + threshold
        elif intensity == 2:
            cornerStrength = pointMin - max(contNine) + threshold

        #print("corner", pointInt, pointMax, pointMin, intensity, points)
        keypoints.append([point, intensity, cornerStrength])
        break
return keypoints
```

If the corner is found, then it again checks if the corner was lighter or darker, and then computes the **corner strength** using the case shown in the design. This is used in the non-maximal suppression. Then the corner, if it was lighter or darker and the strength are added to the keypoints.

Because the orientation only has to do with how the descriptors are calculated, I don't have to add any other code other than finding the keypoints here. I will add the intensity centroid in the BRIEF Description process.

The full code for detecting the corner, without the starting calculations for limits and getting calling the circle16Pixel function.

Figure 5.11: Detecting Corner Code

```
for i in range(len(points)):
    #print()
    intensity = 0 # 1 == darker, 2 == brighter, 0 = Similar
    corner = True
    #print(points[i])
    if points[i] >= pointMax:
        intensity = 1
    elif points[i] <= pointMin:
        intensity = 2
    else:
        intensity = 0
        #print("next")
        continue

    contNine[0] = points[i]
    for j in range(1, N): #check next 8 pixels
        #print("comp:", points[((i + j) % len(points))])
        contNine[j] = points[((i + j) % len(points))]
        if intensity == 1: #if lighter
            if points[((i + j) % len(points))] < pointMax:
                #print("yes")
                corner = False #is not corner if not 9 contigious
                break
        elif intensity == 2: # if darker
            if points[((i + j) % len(points))] > pointMin:
                #print("no")
                corner = False
                break

    if corner:
        if intensity == 1:
            cornerStrength = min(contNine) - pointMax + threshold
        elif intensity == 2:
            cornerStrength = pointMin - max(contNine) + threshold

        #print("corner", pointInt, pointMax, pointMin, intensity, points)
        keypoints.append([point, intensity, cornerStrength])
        break
return keypoints
```

With the findNContPixels function now made, I create another function to call this upon all the points in the image.

Figure 5.12: Run FAST Detector on all Points

```
def findFeatures(image, n, threshold):
    keypoints = []
    st = time.time()
    for row in range(3, len(image) - 4):
        for col in range(3, len(image[0]) - 4):
                findNContPixels(image, keypoints, (row, col), n, threshold)
    print("Find feature time: ",time.time() - st)
    return keypoints
```

Here I can initialize the keypoints list, and create a start point and end points so I can track how long the algorithm takes to run on all the points. As the detector requires a circle of surrounding pixels with a radius of 3, I have to check all pixels within 3 of the border of the image, otherwise there would be indexing errors.

5.3.2 Nonmaximal Supression

As outlined in the methodology, there is no way for the corner detector to know if a keypoint is next/close to another keypoint that has already been found. So to counteract this, the algorithm uses nonmaximal suppression. And as I described in the design from [10], I use a corner matrix with the corner strengths as the values, 0 if no keypoint was found at that point and use the 3 x 3 mask to check if there is anything with a higher strength around it.

Figure 5.13: Start of Nonmaximal Suppression Code

```
def nonMaximalSuppression(image, keypoints):
    suppressedKeypoints = []
    st = time.time()
    corners = numpy.zeros((len(image), len(image[0])))
    for i in range(len(keypoints)):
        corners[keypoints[i][0][0]][keypoints[i][0][1]] = keypoints[i][2]
```

I start by having the image and the keypoints of that image passed into the function and I set up a new list of keypoints, *suppressedKeypoints* which will be the reduced subset of the corner keypoints. I also initialize a new array with the same size as the image with 0s using Numpy, and then fill that matrix with the corner strengths from the keypoints, this will be the corner matrix. Using an Numpy.array allows for very quick editting and viewing because of the fixed size, as well as being able to create the corner matrix is very few lines of code.

There is a similar approach I use in the nonmaximal suppression as in the corner detector, as the keypoints will always be within 3 pixels of the border, I don't have to check the edges. Shown in **5.14**, you can see that in the for loop I check to see if the value isn't a 0, which would indicate a non-corner. If it is a corner, then it gets the neighbours of the corner pixel, which is done in a separate function, shown in **5.15**, and sets a boolean variable, *maxStrength* to True. This variable in the same way as the boolean value *corner* in the corner detector, where I try to find if the corner that is selected is not the max, rather than is the max strength. I then check the neighbours to see if there is a corner that has a higher strength. If no higher strength is found, then the corner gets added to the *suppressedKeypoints*, if not, then that value is set to 0 and the next corner is run.

Figure 5.14: Preforming the Mask on the Corner Matrix

```
#preform non-maximal suppression on the features
for row in range(3, len(image) - 4):
    for col in range(3, len(image[0]) - 4):

        if corners[row][col] != 0:
            neighbours = getNeighbours(corners, (row, col))
            maxStrength = True
            x = False

            for neighbour in neighbours:
                if neighbour >= corners[row][col]:
                    maxStrength = False
                    corners[row][col] = 0
                    break
            if maxStrength:
                suppressedKeypoints.append((row, col))


print("Non Maximal Suppression time:", time.time() - st)
return suppressedKeypoints
```

I also have a timer for when the algorithm starts, and then print the time at the end, so that I can track how long the algorithm takes on different images.

Figure 5.15: Getting the Neighbours of a Point

```
def getNeighbours(image, point):
    (row, col) = point
    neighbours = numpy.array(range(8))
    neighbours[0] = image[row - 1][col - 1]
    neighbours[1] = image[row - 1][col]
    neighbours[2] = image[row - 1][col + 1]
    neighbours[3] = image[row][col + 1]
    neighbours[4] = image[row + 1][col + 1]
    neighbours[5] = image[row + 1][col]
    neighbours[6] = image[row + 1][col - 1]
    neighbours[7] = image[row][col - 1]
    return neighbours
```

The function getNeighbours is very simple and just goes around the pixel from top left clockwise and returns a Numpy.array of size 8 with the 8-neighbourhood as the values.

5.4 Visualisation of FAST

The visualisation of the FAST keypoints is important so that I, or a user, can see the results of the program. The skimage.io package has a function io.imshow, which will draw the image, but this is not useful without having drawn the keypoints onto the image. So I use the skimage.draw package to be able to draw circles onto the image to be able to highlight the keypoints.

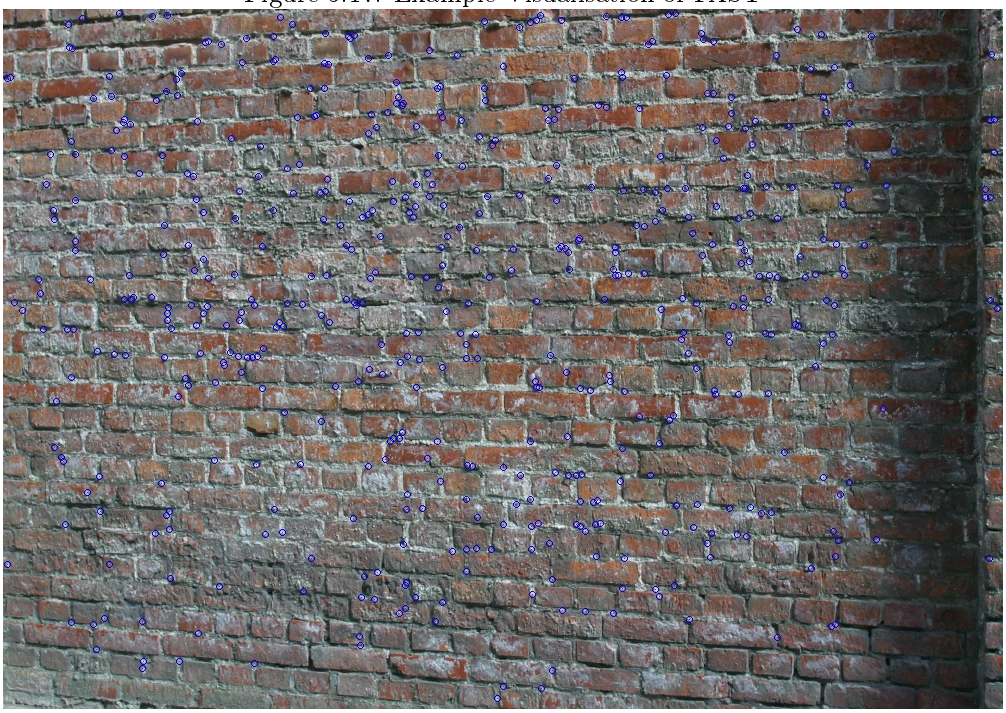
Figure 5.16: Drawing result of FAST Corner Detector

```
def drawFeatures(img, keypoints):
    colourImage = color.gray2rgb(img)
    for keypoint in keypoints:
        #print(keypoint)
        rr, cc = draw.circle_perimeter(keypoint[0], keypoint[1], 3)
        colourImage[rr,cc] = [0,0,250]
    for keypoint in keypoints:
        colourImage[keypoint[0]][keypoint[1]] = [255,0,0]
    io.imshow(numpy.uint8(colourImage))
```

I first set the image back to be colour as then I can make the keypoints easily visible. Then for each keypoint I do two things, first I create a circle around the image where the surrounding 16 pixels would be in full blue RGB, and second, I place a full red RGB dot where the keypoint is. Then I use skimage.io.imshow whilst also making the image Numpy.uint8 type as that gives the best image look.

The output of this with the image being wallOne (figure 5.3), and the threshold value set to 90. You can see all the keypoints have red points, and a circle in blue to highlight them. Shown in **figure 5.16**. Changing the threshold value will give different results, I will explore that in the testing section.

Figure 5.17: Example Visualisation of FAST



5.5 BRIEF Descriptors

After getting the keypoints from the FAST system, I need to give each keypoint a description using the BRIEF binary descriptor described in the methodology and design. The BRIEF process requires a blurred image, using Gaussian blur (GB). I created my own GB function but it turned out to be much slower than just using the GB from the skimage.filters package, so instead I will be using that instead.

Figure 5.18: Start of the BRIEF Function

```
def briefDescriptors(image, keypoints, patchSize=31, matching = False, testPairs = []):
    image = imageToGrey(image)
    st = time.time()
    #blurredImage = guassianBlur(image)
    sigma = numpy.sqrt(1/25 * patchSize ** 2)
    blurredImage = filters.gaussian(image, sigma, truncate=4.5) #is just faster than the
    blurredImage = exposure.rescale_intensity(blurredImage, out_range=(0,255))
    blurredImage = numpy.int64(numpy.uint8(blurredImage))
```

For the input parameters, there is the image that is being used, with the keypoints done by FAST. The patch size is assumed to be 31, as that is what is used in [15]. For when matching is being done, the same test pairs have to be used, otherwise you wont get any matches due to how the test pairs are found. So both *matching* and *testPairs* will be used in the matching algorithm.

Again I use the imageToGrey function to make sure there is only 1 intensity channel, which makes it much easier to work with the image, and get a start time to track how long the algorithm takes. You can see that I initially used my own gaussianBlur function, but later changed to the skimage, to save time on the algorithm. This also gives back a Numpy.float64 image with scale 0-1, similarly in the imageToGrey function, I rescale the image, and change the type to Numpy.int64. The sigma value is also calculated to be $\sigma^2 = \frac{1}{25}S^2$, as described in the methodology.

The sigma and truncate in the filters.gaussian function as used to change the sigma values that will be used in the Gaussian distribution, and truncate is used to detirmine the size of the mask used to blur the image. Described in the methodology, the size of the mask was to be 9x9, so to get this, I set the truncate value to 4.5.

Figure 5.19: Getting the Test Pairs

```
def getTestPairs(patchsize, sigma):
    maxPatch = numpy.int64(patchsize / 2)
    testPairs = numpy.around(numpy.random.normal(0, sigma, (256, 4)))
    testPairs = numpy.int64(testPairs)
    for i in range(len(testPairs)):
        for j in range(0,4):
            if testPairs[i][j] > maxPatch:
                testPairs[i][j] = maxPatch
            elif testPairs[i][j] < -maxPatch:
                testPairs[i][j] = -maxPatch
    return testPairs
```

Before I can create the descriptors, I need to find the pairs of pixels that will be tested using the intensity test to be able to create the bit-string. The method that was chosen in the paper was to use the Gaussian distribution (which is also called the normal distribution) to come up with random pairs of n_d pixels, with n_d being 256 in this case. This can be done using the Numpy.random.normal function, which returns a set

size of array, filled with normally distributed values. I set the size I want to be (256, 4) which gives an array of 256 tuples each of length 4, these 4 values (x_1, y_1, x_2, y_2), will be the pairs I use for the test pairs.

I also have to make sure that the max value that the values is half that of the patch size, this is because the sigma value used can return values that index to outside of the patch. For this, I bound these values to the edge of the patch.

Figure 5.20: Calling testPairs Function

```
if not matching:
    testPairs = getTestPairs(patchSize, sigma)
```

As said before, during matching the testPairs will be passed in due to having to use the same test pairs on different images. So I check the matching parameter and get a new set of test pairs depending if I am matching image.

Figure 5.21: Making the Descriptors, Part 1

```
maxPairRow = len(image)
maxPairCol = len(image[0])
for i in range(0, len(keypoints)):
    (row, col) = keypoints[i]

    #gives rounded 256, 2 for the patch tests
    #print(testPairs[0])
    j = 0
    bitStringMpz = xmpz(0)
    for pair in testPairs:
        xPair = [row + pair[0], col + pair[1]]
        yPair = [row + pair[2], col + pair[3]]
```

With all the previous done, I can now start to construct the descriptors, I make variables to find the max the values can be as the patch size can go over the bounds of the image. Then for each of the keypoints that were found, I create a bit-string using the gmpy2.xmpz package then for each of the test pairs, I find where they should be in the image relative to the keypoint. To know which value of the bit string is being calculated, I initialize a variable $j = 0$, and this will increase by 1 per test pair.

But because the test pair values can be outside the bounds of the image, I have to check if each of the values is above the max values, or below 0. If either of these are the case, the values are then bounded to the edge of the image, which is either the max value or 0, respectively. This is shown in **5.22**. Each if/elif statement is checking each point if its above the max, or below 0, and setting the value accordingly.

With the test pairs found and bounded to the edge of the image if needed, I can then get the value of that bit by using the briefTest, shown in **5.24**. The intensities are found using the test pairs coordinates on the blurred image. The value of the bit, either 1 or 0, is then place onto the j^{th} bit, then the j is increased by 1. After all the test pairs have run, there will be a bit string of length 256. This then gets added to the keypoint list along with where that keypoint was, shown in **5.23**.

When all the keypoints have received a BRIEF description, the keypoints is returned as well as the test pairs used for the descriptors. This can be used to match images against it.

Figure 5.22: Making the Descriptors, Part 2

```
if xPair[0] < 0:
    xPair[0] = 0
elif xPair[0] >= maxPairRow:
    xPair[0] = maxPairRow - 1
if xPair[1] < 0:
    xPair[1] = 0
elif xPair[1] >= maxPairCol:
    xPair[1] = maxPairCol - 1

if yPair[0] < 0:
    yPair[0] = 0
elif yPair[0] >= maxPairRow:
    yPair[0] = maxPairRow - 1
if yPair[1] < 0:
    yPair[1] = 0
elif yPair[1] >= maxPairCol:
    yPair[1] = maxPairCol - 1
```

Figure 5.23: Making the Descriptors, Part 3

```
        #print(blurredImage[xPair[0]][xPair[1]])
        bit = briefTest(blurredImage[xPair[0]][xPair[1]], blurredImage[yPair[0]][yPair[1]])
        #bitString = bitString + bit
        bitStringMpz[j] = bit
        j = j + 1
    #print(bitString)
    #print(j)
    keypoints[i] = (row, col, bitStringMpz)
    #keypoints[i] = (row, col, bitString)
print("BRIEF time: ",time.time() - st)
return (keypoints, testPairs)
```

Figure 5.24: BRIEF Intensity Test

```
def briefTest(intensityX, intensityY):
    if intensityY > intensityX:
        return 1
    else:
        return 0
```

Due to the timing constraints of the project, I didn't get around to adding the rotational invariances to the BRIEF features. This will be a problem if I were to run the system with images that have large rotations. As said in the original BRIEF features paper [4], the system can still preform well with slight rotation, but as the angle gets larger, the system will preform worse.

5.6 Matching System

Matching two images requires to run the FAST detector and the BRIEF descriptor on both the images with the same parameters entered for both. As said in the BRIEF implementation, I have adding a test pair parameter for the BRIEF descriptions so that the matching can take place. Afterwards I will use a greedy nearest neighbour algorithm to find the matches that come out of the two images.

Figure 5.25: Code for Matching Images, Part 1

```
def matching(img1, img2, patchSize=31, limit=60, threshold=30):
    sigma = numpy.sqrt(1/25 * patchSize ** 2)
    pairs = getTestPairs(patchSize, sigma)

    matches = [] #tuples of two features that match

    imageOne = imageToGrey(img1) #image with whole values
    imageTwo = imageToGrey(img2) #image with whole values
```

For the input parameters I have the two images, the patch size and the threshold, which are all used in the FAST and BRIEF algorithms, but the *limit* parameter is used for the matching process, this will be used to make sure that the matches have an upper bound to what they can get matched to.

Similar to the BRIEF descriptors, I work out the sigma values used for the algorithms using the patch size and getting the test pairs that will be used. As well as initialize the matches list and making sure the images are greyscale.

Figure 5.26: Code for Matching Images, Part 2

```
imageOneKeypoints = getFeatures(imageOne, threshold=threshold)
imageTwoKeypoints = getFeatures(imageTwo, threshold=threshold)
imageOneDiscriptors = briefDescriptors(imageOne, imageOneKeypoints, patchSize=patchSize, matching=True, testPairs=pairs)
imageTwoDiscriptors = briefDescriptors(imageTwo, imageTwoKeypoints, patchSize=patchSize, matching=True, testPairs=pairs)
```

As I previously said, I have to run both images through the FAST and BRIEF systems, which is what is shown in **5.26**. The arguments for the BRIEF descriptors are all filled in with the same test pars as well as having *matching* set to True, so the code knows to use the same test pairs to allow matching.

Figure 5.27: Code for Matching Images, Part 3

```
#Greedy Nearest Neighbour with a limit
st = time.time()
for one in range(0,len(imageOneDiscriptors[0])):
    match = False
    currentDist = limit + 1
    for two in range(0, len(imageTwoDiscriptors[0])):
        distance = hamdist(imageOneDiscriptors[0][one][2], imageTwoDiscriptors[0][two][2])
        #print(currentDist, distance, limit)
        if distance < limit:
            #print("match")
            match = True
            if distance < currentDist:
                #print("better match")
                currentDist = distance
                currentMatch = (imageTwoDiscriptors[0][two], two)
    if match:
        matches.append((imageOneDiscriptors[0][one], currentMatch[0]))
```

Now I begin to match the features in both the images together with a nearest neighbour greedy search. With again a start time set so I can track how long the algorithm takes.

I use a for loop to go through all feature in the first image and set the match boolean to False, the reason this is different than what I used before is that I can't assume a match has been found. I create a variable called *currentDist*, as I will be working with the hamming distance as the comparison to find matches, I will need to save what the lowest distance that has been found for the current matching pair, this is where I use *currentDist*.

Then I set another for loop up for the second image features, then for each feature in the second image, I check the distance between the image one feature and the image two feature. If that distance is lower than the limit that has been set, then a match has been found. The program then looks to see if that match is lower than the current match that has been found, byt checking the distance against current distance. If the two features have a better match, then the current distance and the current match are updated.

Once all the features in the second image have been checked against the image one feature. The program checks if a match was found, and appends it to the list if so.

The reason that this is a greedy search is that I check all features against eachother. There is also no method of checking if there is alread a match for some of the second image features, so there can be matches with more than one image one feature with the same image two feature.

Figure 5.28: Code for Matching Images, Part 4

```
print("Matching Time:", time.time() - st)
print("Matches Found:", len(matches))
drawMatchFeatures(img1, img2, imageOneKeypoints, imageTwoKeypoints, matches)
return matches
```

Once all the features have tried to be matched together, the time taken and how many matches are printed and the visualisation is run on the two images with their keypoints and the matches.

5.7 Visualisation of Matching

The visualisation of the matching is very useful to see and you can easily see if the matching are working as intended and actually matching similar features together. I use similar code as the visualisation of the FAST keypoints, to show both the image's keypoints, and then draw lines between the features that have matched.

Figure 5.29: Visualisation of the Matching Features, Part 1

```
def drawMatchFeatures(imageOne, imageTwo, imageOneKeypoints, imgTwoKeypoints, matches):
    matchedImage, width = concatImages(imageOne, imageTwo)
    print(len(imgTwoKeypoints))
```

For the inputs, I need the two images, with their keypoints and the matches that have been found bewteen them. The skimage package doesnt come with any functions to concatinate two images together, so I had to write my own function, which I called *concatImages*. This returns the images next to eachother, and the width to shift the second images keypoints by.

The concatination of the two images required finding the max size of the width and the height of the two images. I can then initalize a numpy.array with size of the max height and 2 times the max width to allow room for both the images. Then I have to check which parts of the images have to be increased by to get to the max width/height. For example, the width of wallTwo is 680, but the width of wallOne is 700, so

Figure 5.30: Padding the Images

```
def concatImages(imgOne, imgTwo):
    imgHeight = max(len(imgOne), len(imgTwo))
    imgWidth = max(len(imgOne[0]), len(imgTwo[0]))

    combinedImage = numpy.zeros((imgHeight, imgWidth*2, 3))
    #print(imgHeight, imgWidth, "\n", len(imgOne), len(imgOne[0]),"

    if len(imgOne) < imgHeight:
        heightPad = imgHeight - len(imgOne)
        imgOne = numpy.pad(imgOne, [(0, heightPad), (0,0), (0,0)])
    if len(imgOne[0]) < imgWidth:
        widthPad = imgWidth - len(imgOne[0])
        imgOne = numpy.pad(imgOne, [(0, 0), (0,widthPad), (0,0)])

    if len(imgTwo) < imgHeight:
        heightPad = imgHeight - len(imgTwo)
        imgTwo = numpy.pad(imgTwo, [(0, heightPad), (0,0), (0,0)])
    if len(imgTwo[0]) < imgWidth:
        widthPad = imgWidth - len(imgTwo[0])
        imgTwo = numpy.pad(imgTwo, [(0, 0), (0,widthPad), (0,0)])
```

wallTwo needs to be padded with 20 pixels to be the same size. This happens for both the widths and heights for each image.

Figure 5.31: Combine the Images

```
combinedImage[numpy.ix_(range(0,imgHeight), range(0,imgWidth))] = imgOne[numpy.ix_(range(0,imgHeight), range(0,imgWidth))]
combinedImage[numpy.ix_(range(0,imgHeight), range(imgWidth,imgWidth*2))] = imgTwo[numpy.ix_(range(0,imgHeight), range(0,imgWidth))]

combinedImage = numpy.uint8(combinedImage)
return combinedImage, imgWidth
```

With the padding done, I can place the two images in the *combinedImage* array. I use the numpy.ix_ function which allows me to select a region of the array to be able to change, so I select the first region of the array where I place the first image, and then offset the placement of the second image by the max width. This then get turned into a Numpy.uint8 array ready to be displayed/saved.

In **5.32** I show the code for the drawing on the images. After getting the width of the images and the combined image, I draw the keypoints on the two images using the same code that was in the visualisation of the FAST keypoints, but for the second image keypoints I have to shift the circles and points by the width of the image so that they appear in the right place.

For the match lines, I have to get the start and end point, I get this from the matches array that was passed in, but again I have to make sure that the end row is shifted by the width so that it appears on the second image. After finding the points, I can use the skimage.draw package to use the line function and draw and bright green line between the features.

After the keypoints and the lines have been drawn on, I save the image to file as a .jpg image for quality and display the image in the IDE.

I show an example of the matching algorithm visualisation in **5.34**, in this particular example, I use wallTwo as the first image and wallOne as the second. The threshold was set to 60, and the limit was set to 15. I will explore how the limit and threshold affect the matching in the testing section.

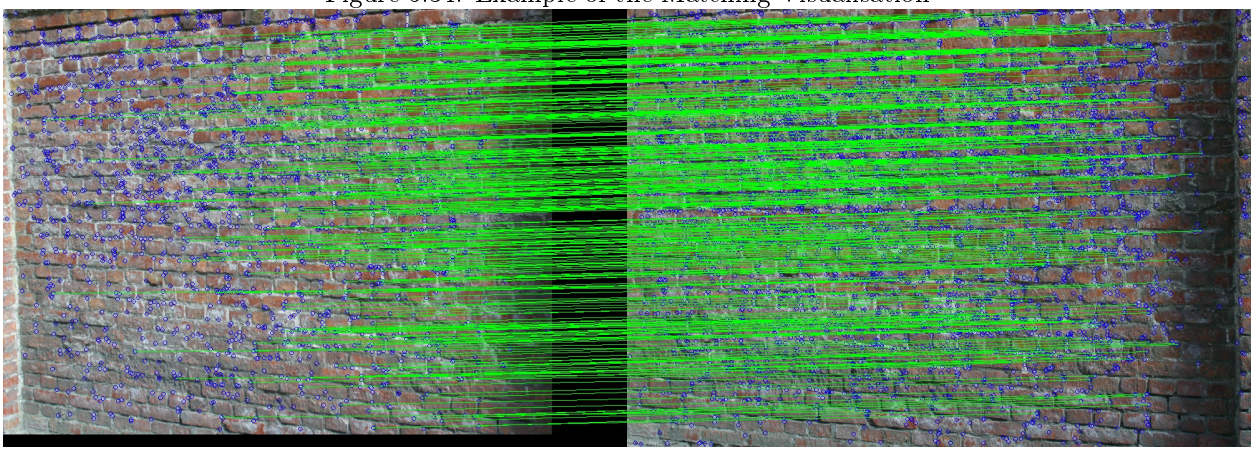
Figure 5.32: Visualisation of the Matching Features, Part 2

```
for keypoint in imageOneKeypoints:
    #print(keypoint)
    rr, cc = draw.circle_perimeter(keypoint[0], keypoint[1], 3)
    matchedImage[rr,cc] = [0,0,250]
    matchedImage[keypoint[0]][keypoint[1]] = [255,0,0]

for keypoint in imgTwoKeypoints:
    #print(keypoint)
    rr, cc = draw.circle_perimeter(keypoint[0], (keypoint[1]+width), 3)
    matchedImage[rr,cc] = [0,0,250]
    matchedImage[keypoint[0]][keypoint[1]+width] = [255,0,0]


for i in range(0, len(matches)):
    startRow, startCol, discriptor = matches[i][0]
    endRow, endCol, discriptor = matches[i][1]
    endCol = endCol + width
    rr, cc = draw.line(startRow, startCol, endRow, endCol)
    matchedImage[rr,cc] = [0,255,0]
```

Figure 5.33: Visualisation of the Matching Features, Part 3

```
io.imsave("matchings2.jpg", matchedImage, quality=100)
io.imshow(numpy.uint8(matchedImage))
```

Figure 5.34: Example of the Matching Visualisation



6. Testing and Validation

6.1 Testing

In the testing and validation section I will be exploring how the different arguments I give the functions that I created affect the end results, as well as the how fast the algorithms are. The point will be to see how well I achieved my aims and goals stated in the introductions.

6.1.1 Datasets and Images

To be able to preform the tests I need a dataset of images that I can use to see how well the algorithms preforms. I already introduced two images of a dataset of the same wall at different angels, to this I add 4 more pictures of the wall which I can preform matching on. These are shown in **6.1** to **6.6** which shows the 6 images of the wall.

Figure 6.1: wallOne

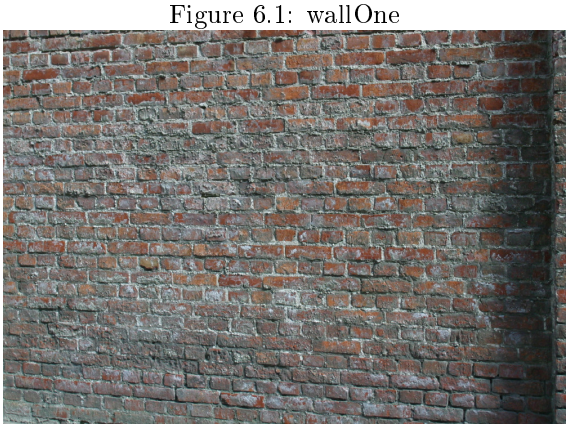
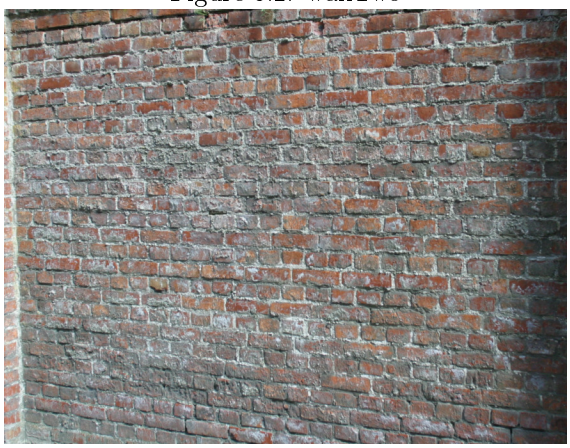


Figure 6.2: wallTwo



‘

Figure 6.3: wallThree

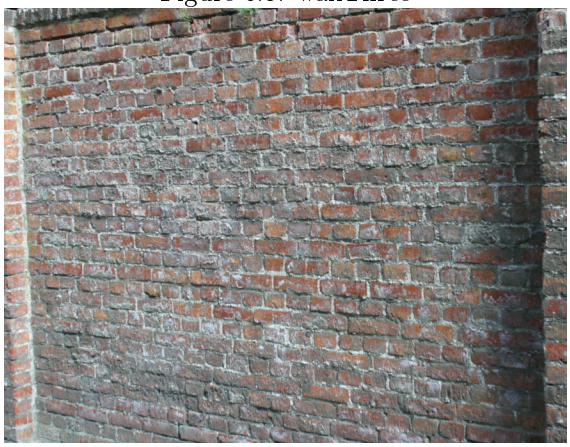
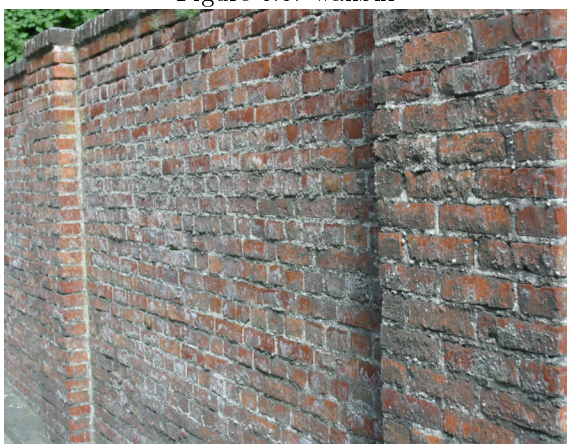


Figure 6.4: wallFour

Figure 6.5: wallFive

Figure 6.6: wallSix



6.1.2 Testing the System

Recognition Rates

The first goal I will looking at is how well the implementation works in terms of recognition rates. For the purpose of this I will be defining the recognition rate similarly to how the BRIEF paper [4], where I run the algorithm and pick N points from both images, and run the matching algorithm on the descriptors. As I am using nearest neighbour for the matching system, the recognition rate will be how many correct matches that have been found between the two images divided by the number of points. $\frac{r_c}{N}$, where r_c is the number of correct matches.

This process will be preformed on the wall dataset and each wall will be matched against wallOne. Then choosing the value of N to be the number of features that are found on the image with the least features.

I will be comparing the results I get to those that were found in the BRIEF paper [4], to see how well my system preforms against another of the same. I will not trying to prove that a BRIEF system is better than a different types of descriptors, such as SIFT or SURF.

To be able to control some of the test, I need to choose a value for the threshold and the limit as these affect the amount of features that are found and how close they can be to be a match. For this I will

choose the threshold to be 75, and the limit to be 10. With these arguments I preform the algorithm on the images:

Table 6.1: Table of Recognition Rates

Images		Features Found					
Image One	Image Two	One	Two	Matches	r_c	Correct Rate	Recognition
wallOne	wallTwo	1537	603	150	141	0.94	0.233
wallOne	wallThree	1537	545	79	74	0.936708861	0.135779817
wallOne	wallFour	1537	624	39	27	0.692307692	0.043269231
wallOne	wallFive	1537	630	37	19	0.513513514	0.03015873
wallOne	wallSix	1537	708	15	1	0.066666667	0.001412429

As you can see from this table, I preformed the matching algorithm on the different images. Each time you can see that the amount of features for wallOne do not change as the same threshold is used to find the keypoints, so the same features are found, each time the description changes but this shouldn't affect the end result too much as both images use those test pairs.

You can see that for the matching between wallOne, wallTwo and the matching of wallOne, wallThree give a good percentage of correct matches for those that were found, this is because of the images similarity to each other. The more similar the image, the more of the same features that will be pulled out.

Because of the image of the wall being rotated, if I had implemented the rotational invariance, the correct rate and recognition rates would've been higher for the higher number wall image. As this is not the case you can see that as the wall gets rotated more, the correct rate and recognition rate begin to plummet. This is largely because of how I have set up the algorithm, because I am using a hard limit to the features matching potential, the matches that would come from being above the limit never get found. The solution to this would be to create a system that could detect whether a match is actually true or not, I will explain one such a system later on.

You can compare the results to those that were found in [4], in the paper they get much higher recognition rates than what I achieved here. This could be from lack of understanding of how they preformed their tests, or because of the slight differences on how the algorithms were coded. There is still a trend that is shown in both, the recognition rate and the correct rate both decrease rapidly for the higher number wall image, where wall's one, two and three matching give very high, but the matching between wall 1 and 6 do not. Both trends come from the rotational varience of the wall's images and could be somewhat fixed by implementing the ORB's oriented BRIEF system.

For the matches that were found, the correct rate is still high for all but the last matching, which shows success on trying to match images together when the images are similar. But when the images get too dissimilar, then the matches start to fail and using a nearest neighbour greedy search with a limit is not the way to be able to find these more complex matches. This shows that the system could be used to match and detect objects to be used for more complex methods such as classification.

Speed of the Algorithms

As the project is about feature extraction, the speed of the algorithm being increased is very useful. Not as crucial as the accuracy but the techniques used here can reach real-time speeds, meaning the could be used on videos. I will be using the same threshold and limit as before, with 75 and 10, respectively.

As you can see from the table below, with the FAST algorithm I coded, the system is not very fast, it takes multiple seconds to find the descriptors for an image of this size. So instantly we know that the system wouldn't work on videos, as my project was on just feature extraction this is okay.

Table 6.2: Table of Algorithm Times

Image	FAST time /s	NonMax Time /s	BRIEF time /s
wallOne	4.915	0.279	1.282
wallTwo	4.277	0.21	0.505
wallThree	4.404	0.212	0.639
wallFour	4.427	0.21	0.717
wallFive	4.423	0.211	0.55
wallSix	4.576	0.221	0.684
Average	4.503666667	0.223833333	0.7295

But because the techniques can reach those type of speeds, it brings up why mine is no where near that speed. This is likely due to how I coded it and what I used to code the algorithms, python as a language isn't the fastest out there for this type of problem, C would've been a much better candidate, but I chose python out of comfort as I already knew it well and the libraries that go along with it. I tired to cut down the time by using the Gmpy2 package for the BRIEF descriptors which should have lowered the time. And the non-maximal suppression is around a constant 0.2s, which is promising and likely could be improved just by using different libraries or another language.

This shows that even though my system works for similar images, there is still much more improvements to be made in the coding, using better libraries and/or using better coding practices may increase the speed to that of real time.

6.1.3 Effect of Threshold and Limit Values

As a base understanding we know the values directly affect the amount of features and the amount of matches that the threshold and the limit give respectively. I will be using the image wallOne to see how the 2 affect the results. I start with the threshold being very low at 15, which affects the FAST algorithm so I will just be running that. Running the algorithm with such a low threshold for corners will give back a lot of features, which you can see here:

Figure 6.7: Running wallOne through the FAST system, $t = 15$

```
x = getFeatures(io.imread("wall/img1.ppm"), 15)
Find feature time:  8.89325475692749
Features found:  130002
Non Maximal Suppression time: 2.107623815536499
Final Features found:  45065
```

Having the threshold set to 15 gives back 130000 features before the non-maximal suppression and then 45000 after, this is basically unusable as there are too many features that likely mean nothing but just because the threshold was so low get picked up as a corner. This could be just picking up colour discrepancies in the image, so being this low is not helpful for the system.

Figure 6.8: Running wallOne through the FAST system, $t = 30$

```
x = getFeatures(io.imread("wall/img1.ppm"), 30)
Find feature time:  6.163639068603516
Features found:  48432
Non Maximal Suppression time: 0.9574663639068604
Final Features found:  22103
```

When raising the threshold to double, you get roughly a 1/3 of the features, and then down to 20000 after NMS. This cuts most of out the colour discrepancies, but is still giving back a larger number of features that are either useless or are too close to another feature. This is because NMS only gets rid of the closest of the of the features that are near to eachother, if they are a few pixels apart they could still be describing the same feature. So raising the threshold more is needed.

Figure 6.9: Running wallOne through the FAST system, $t = 45$

```
x = getFeatures(io.imread("wall/img1.ppm"), 45)
Find feature time:  5.481207609176636
Features found:  18000
Non Maximal Suppression time: 0.5196809768676758
Final Features found:  10061
```

Getting to $t = 45$, we are starting to reach the point where it could be useful. I will also show what the visualisation looks like.

Figure 6.10: Visualisation of wallOne with $t = 45$

Its not the best threshold to use as there are still way too many features being found, but you can now see that most of the features are being bound to the edges of the bricks, which are the corners of most of the image of a brick wall. There are still some issues with some features being close to others and there being a big clump in the centre top right of the image, which shows that the threshold just needs to be higher.

Figure 6.11: Running wallOne through the FAST system, $t = 60$

```
x = getFeatures(io.imread("wall/img1.ppm"), 60)
Find feature time:  5.0469138622283936
Features found:  6458
Non Maximal Suppression time: 0.33047032356262207
Final Features found:  4184
```

Now for $t = 60$, you can see from both the visualisation and the numbers that its getting much more reasonable to work with. Nearly all of the features are isolated, showing that each is describing a different feature. Its not perfect, but the amount of features has reduced to 4184, which is just slightly above what the expected was in the FAST paper [14].

Figure 6.12: Visualisation of wallOne with $t = 60$

I would increase the threshold more just because there is still a few places where the features are a little too close together.

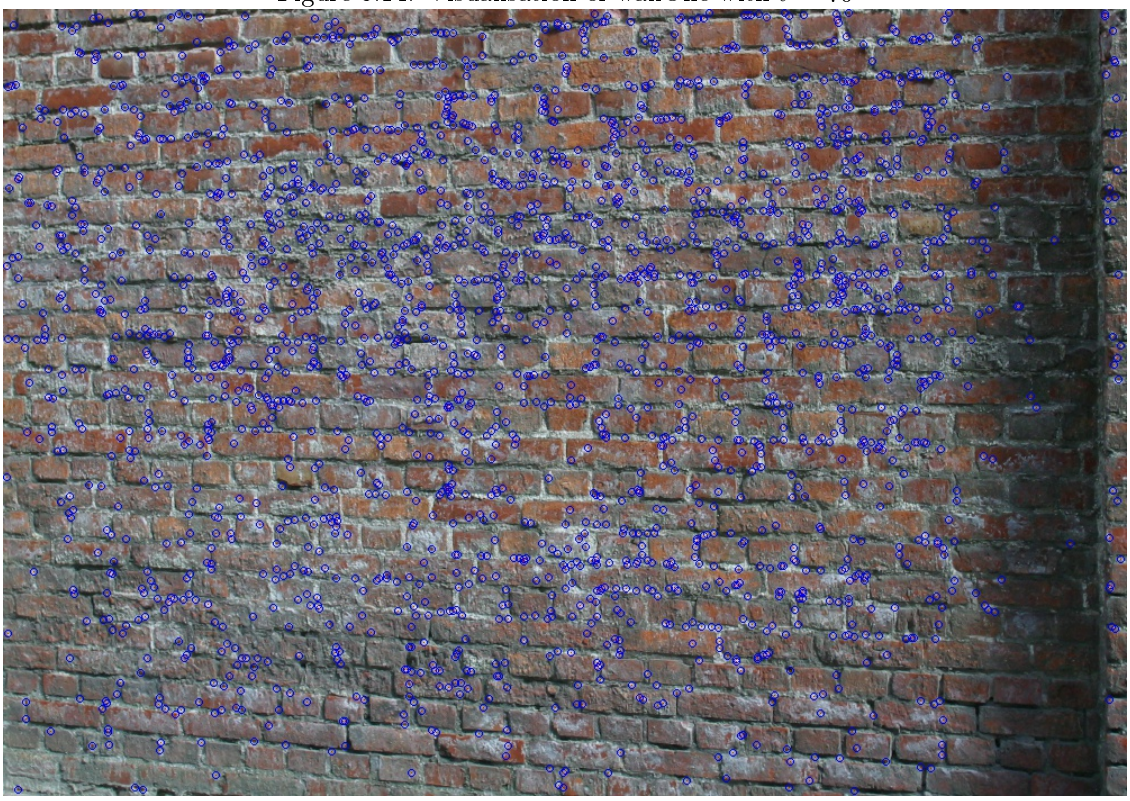
For this, I increase to $t = 70$.

Figure 6.13: Running wallOne through the FAST system, $t = 70$

```
x = getFeatures(io.imread("wall/img1.ppm"), 70)
Find feature time:  5.045929908752441
Features found:  3180
Non Maximal Suppression time: 0.30649852752685547
Final Features found:  2178
```

When I raise the threshold once again to 70, the amount of features is 2178, which is very close to what was being found in the FAST paper, meaning that this is the around the point where the algorithm should be working the best. The speed is still high with 5 seconds to complete the FAST keypoints, but as explained before, its okay as the system still works.

Figure 6.14: Visualisation of wallOne with $t = 70$



The features are not too close to each other and spread out across the whole image, giving a wide range of descriptors for the BRIEF system to use. You can see that at around 70 it gets a good number of features, in the previous section I used $t = 75$, which gave 1500 features. So this puts the best threshold range at about 70-80. When going above these amounts, the amount of features you get become in the 100s, this is not enough to be able to describe an image.

This is only true for an image of a similar contrast, the higher the contrast, the harder it will be for a candidate pixel to be considered a corner, due to the changed in the intensities. So for images with higher contrasts a higher threshold will be needed to get similar results. This is similar for a lower contrast image, where the threshold will need to be lowered.

Changing the Limit

As I've shown the threshold changes how the keypoints of the features are found. But the limit changes how close a match can be to be considered a match. During the code that I run during the implementation to check my code was working, I found that after a point the limit starts to just make ever single feature in the first image have a match to the second. This is due to how I am finding the matches. The nearest neighbour search just tries to find any match it can within the bound of the limit, so having a high limit just causes every point to have a match, even if its not correct. So to counteract that, if you apply a low limit you can reduce the number of incorrect matches, but this also will decrease the number of correct matches too. So you have to find a balance, or use another method of finding the matches.

Looking at the BRIEF paper [4], you can see that during their testing the matches between the walls for those with incorrect matches followed the Gaussian distribution, so when setting the limit to half of the max, which would be 128, you should expect over half of the image to be matched. In reality the number of matches will be the near max due to the contrast of the image being the same all over, and the object of a wall is just comprised of many bricks, so many descriptors will be similar in this case.

I will start with a relativly high limit of 64 to show the effect it has. I will be using the matching between wallTwo and wallOne with a threshold of 80, this is to decrease the number of potential matches and make the results easier to see and comprehend.

Figure 6.15: Visualisation of Matching between wallTwo and wallOne with limit = 64

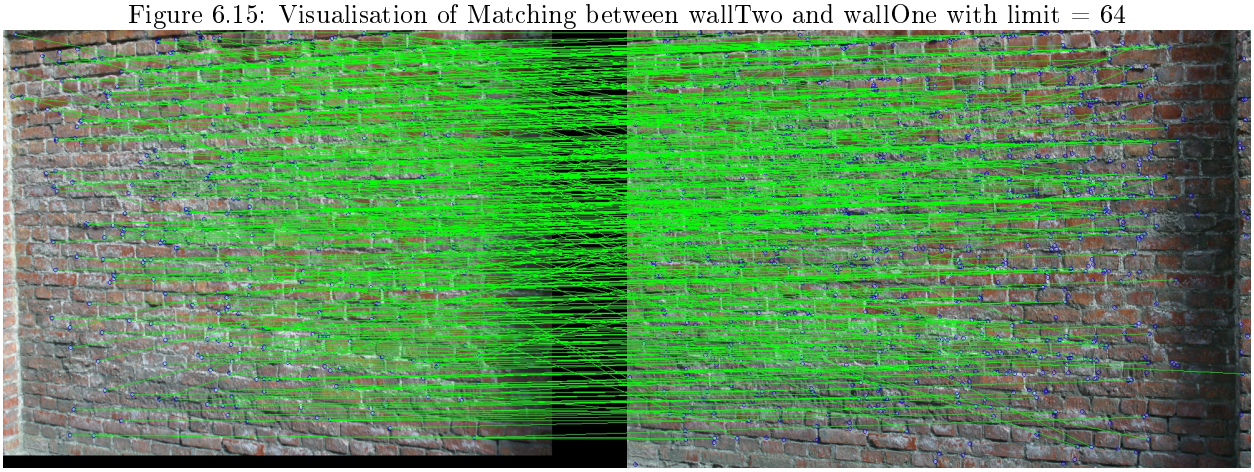


Figure 6.16: Matching wallTwo and wallOne with limit = 64

```
In [30]: x = matching(io.imread("wall/img2.ppm"), io.imread("wall/img1.ppm"),
threshold=80, limit=64)
    ...: Find feature time:  3.970628499984741
    ...: Features found:  569
    ...: Non Maximal Suppression time: 0.19819331169128418
    ...: Final Features found:  403
    ...: Find feature time:  4.805800676345825
    ...: Features found:  1491
    ...: Non Maximal Suppression time: 0.24710392951965332
    ...: Final Features found:  1072
    ...: Matches Found: 403
```

There is 403 features in wallTwo and 1072 features in wallTwo, because I have put wallTwo first the matches maximal will be the amount of features it has, which is 403. The matching system found a match for all of

the feature in the first image, meaning that already at a limit of 64 everything can match. This is due to the similarity of the images, in both subject and contrast. Again from the BRIEF paper, in their testing graphs, when matching wallOne and wallTwo, the distribution of correct matches was already at maximal when the limit is below 50, so it make sense that here we see the maximal number of matches being found.

You can see a trend of the angle of the matching lines, this is showing the correct matches. Any line that is not following the same trend is an incorrect matching. Most of the lines follow this trend so it shows that the system is actually working as intended.

Figure 6.17: Visualisation of Matching between wallTwo and wallOne with limit = 48

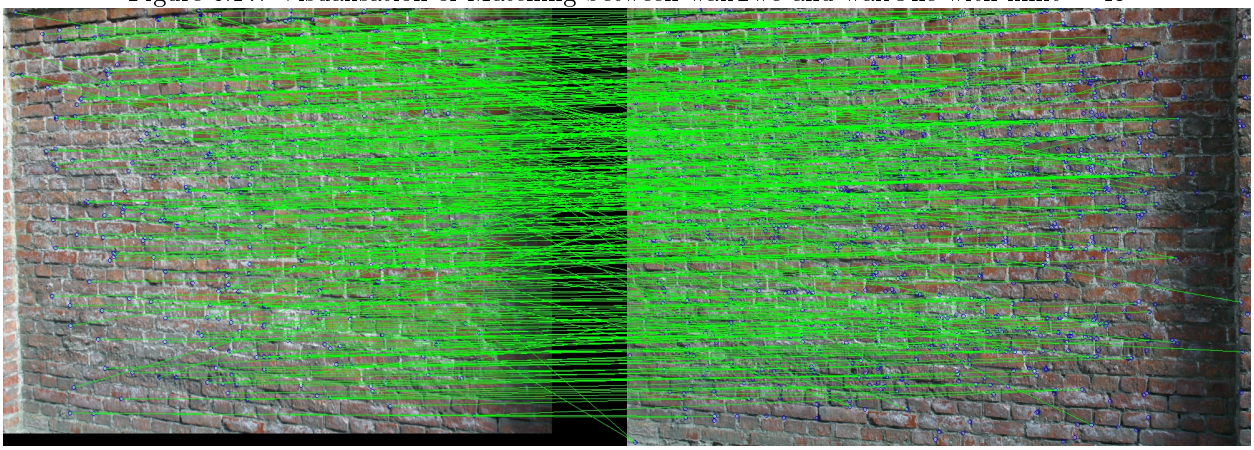


Figure 6.18: Matching wallTwo and wallOne with limit = 48

```
In [31]: x = matching(io.imread("wall/img2.ppm"), io.imread("wall/img1.ppm"),
threshold=80, limit=48)
    ...: wallTwo Features found:  403
    ...: wallOne Features found:  1072
    ...: Matches Found: 395
```

I removed the timing and only showing the feature amount and how many matches were found.

There is still the same number of features being found, but reducing the limit to 48, you can see that now the number of matches reduces to 395. This is likely because it is removing those outliers that only just came in the limit of 64 before. In the images you can't really notice a difference as there is still near maximal matches being found, and there is still a lot of matches that are incorrect as you can see by the lines not following the trend. So the limit is still too high for this type of matching algorithm.

Figure 6.19: Visualisation of Matching between wallTwo and wallOne with limit = 32

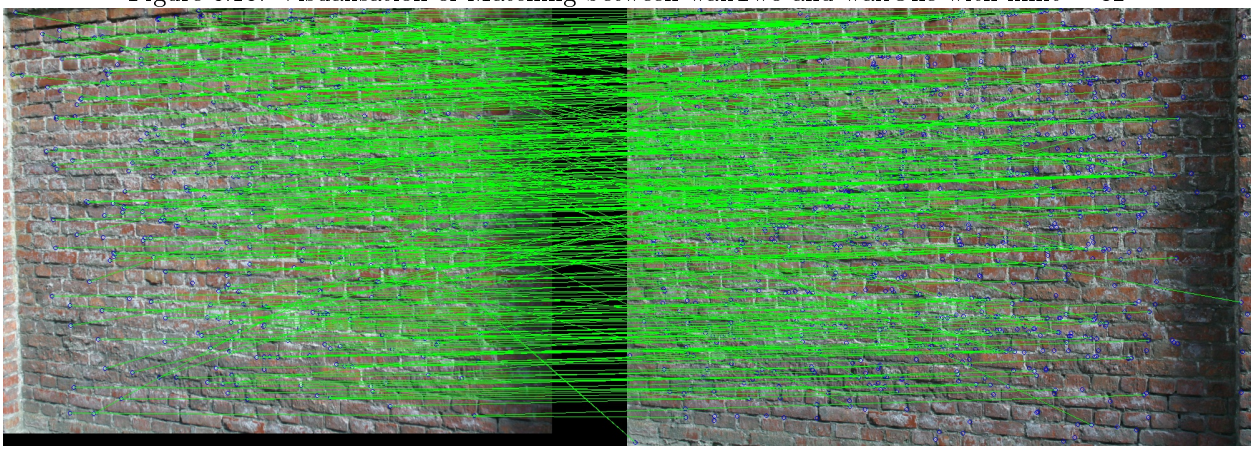


Figure 6.20: Matching wallTwo and wallOne with limit = 32

```
In [33]: x = matching(io.imread("wall/img2.ppm"), io.imread("wall/img1.ppm"),
threshold=80, limit=32)
    ...: wallTwo Features found:  403
    ...: wallOne Features found:  1072
    ...: Matches Found: 362
```

Lowering the limit to 32, there still isn't loads of improvement of the incorrect matches that were found, but there is still a little improvement, the matches went down to 362. The trend of matching lines is still there, indicating that most of the matches were correct, but still some lines that showed incorrect matches. With such little improvement, the limit should be decreased more.

Figure 6.21: Visualisation of Matching between wallTwo and wallOne with limit = 16

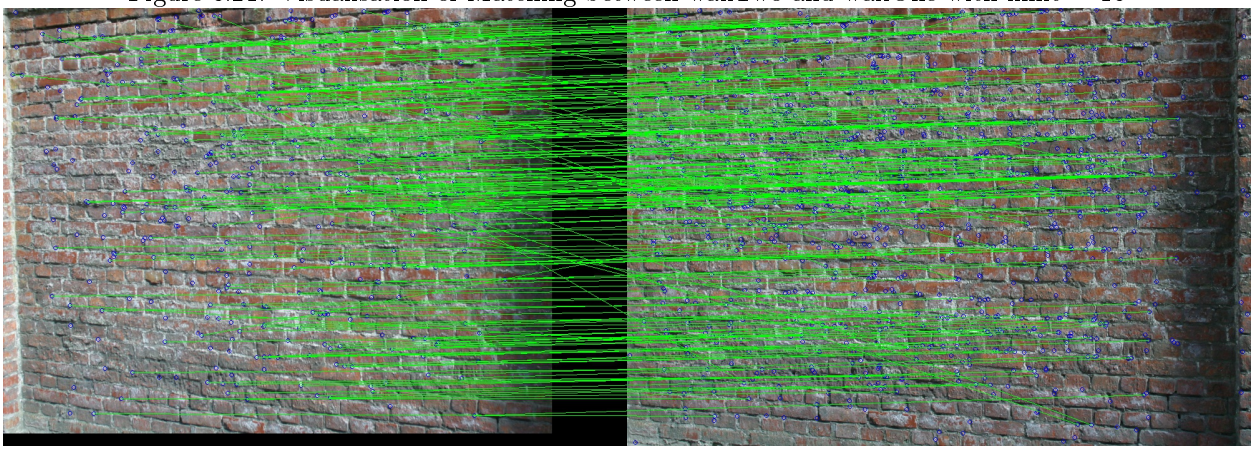


Figure 6.22: Matching wallTwo and wallOne with limit = 16

```
In [33]: x = matching(io.imread("wall/img2.ppm"), io.imread("wall/img1.ppm"),
threshold=80, limit=16)
    ...: wallTwo Features found:  403
    ...: wallOne Features found:  1072
    ...: Matches Found: 212
```

Here is where you see a large jump when reducing the the limit from 32 to 16. The matches decrease to 212, much lower than when the limit was higher. This is because of two reasons, the first being the amount of incorrect matches has decreased, you can see this by the trend of matching lines is still there, but there is less lines that don't follow this trend.

The second is because of the low limit, there are some correct matches that are lost. This is known because of the testing done in the BRIEF paper where the distributions of correct matches is max at about 30. So having a limit lower than this will exclude some correct matches. Because of incorrect matches following a Gaussian distribution, having a limit this low causes the incorrect matches to be minimal. This is shown by the visualisations, you can see that the number of lines that conform and also the lines the don't have reduced. So having a limit at around 16 or lower causes the incorrect matches to be low and most of the matches to be correct.

6.2 What the Testing Shows

As I have shown there is a significant impact on what each of the values means. Changing both to find the best inputs would is a task for another system to find out, or a user testing a bunch of options, which is less feasible if you have a lot of images. For this, the best values I have found are for the threshold to be between 70 and 80, for images of objects in a decent light and similar contrast of that of the wall images, and a limit value of around 16.

The testing also shows that I have successfully created a system that extracts features from an image to be used on an image classifier program.

6.2.1 System for Rejecting False Matches

Earlier I said about a system that I could put in place to find the matches that would be false. The idea is based on angel of which the matching lines follow. As we saw for each of the images there was a trend that

the matching lines follow, this can be used to find where the correct matches are in similar images, such as wallOne and wallTwo. Any lines that would not conform to the trend by a certain margin could be excluded from the matches list, whilst still using the nearest neighbour matching algorithm. This would give better results as most, if not all, of the matches would be correct.

This would also allow for a higher limit, as the matches that are incorrect could be rejected. The threshold could also be decreased to allow for more features and therefore more matches, but the likelihood of similar matches would not decrease accuracy, but further increase the amount of time it takes for the algorithm to be completed.

7. Conclusion

In conclusion, I have created a feature extraction system using the techniques, FAST a corner detection algorithm and BRIEF, a feature descriptor algorithm, which along with orientation components would create the ORB system, but as those were not implemented it doesn't follow the system exactly. The system is able to find features in images based on a threshold (FAST), which value is best between 70 and 80, these features are then fed into the descriptor (BRIEF), and those features can be matched using the hamming distance and a nearest neighbour search with a limit (best values around 16) to find out if the features are similar. It works mostly for similar images, when images become too altered, such as a rotation, the matching system fails to find matches due to the difference in the images, this can somewhat be solved using a higher limit, but at the cost of having more incorrect matches.

This can then go on to be used in classifying software, using the features and the matching with user given labels. You would be able to train a neural network to pick up the similarities of the feature in different images and different clumps of features mean.

In the future I would have implemented the orientation components of the Rotated BRIEF and Oriented FAST, into my system and fully follow the ORB design. This would allow features to be found and matched with rotation invariance. I would also like to go back and change how I coded the FAST system, as it was the first part I coded there is likely some ways that I could improve the algorithm to be up to speed with how it should be. This would also enhance the range of mediums that the system could run on, as real-time feature extraction is a interesting area this type of system can work in.

Bibliography

[1] Octavio Arriaga, Matias Valdenegro-Toro, and Paul Ploger. Real-time convolutional neural networks for emotion and gender classification. *CoRR*, abs/1710.07557, 2017. URL `http://arxiv.org/abs/1710.07557`.

[2] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In Aleš Leonardis, Horst Bischof, and Axel Pinz, editors, *Computer Vision – ECCV 2006*, pages 404–417, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-33833-8.

[3] Marc Berthod, Zoltan Kato, Shan Yu, and Josiane Zerubia. Bayesian image classification using markov random fields. *Image and Vision Computing*, 14(4):285–295, 1996. ISSN 0262-8856. doi: https://doi.org/10.1016/0262-8856(95)01072-6. URL `https://www.sciencedirect.com/science/article/pii/0262885695010726`.

[4] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. In Kostas Daniilidis, Petros Maragos, and Nikos Paragios, editors, *Computer Vision – ECCV 2010*, pages 778–792, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15561-1.

[5] Fangming Chai and Kyoung-Don Kang. Adaptive deep learning for soft real-time image classification. *Technologies*, 9(1), 2021. ISSN 2227-7080. doi: 10.3390/technologies9010020. URL `https://www.mdpi.com/2227-7080/9/1/20`.

[6] Liang-Chi Chiu, Tian-Sheuan Chang, Jiun-Yen Chen, and Nelson Yen-Chung Chang. Fast sift design for real-time visual feature extraction. *IEEE Transactions on Image Processing*, 22(8):3158–3167, 2013. doi: 10.1109/TIP.2013.2259841.

[7] L.C. De Silva and Suen Chun Hui. Real-time facial feature extraction and emotion recognition. In *Fourth International Conference on Information, Communications and Signal Processing, 2003 and the Fourth Pacific Rim Conference on Multimedia. Proceedings of the 2003 Joint*, volume 3, pages 1310–1314 vol.3, 2003. doi: 10.1109/ICICS.2003.1292676.

[8] Delbert Dueck and Brendan J. Frey. Non-metric affinity propagation for unsupervised image categorization. pages 1–8, 2007. doi: 10.1109/ICCV.2007.4408853.

[9] Yousun Kang, Hiroshi Nagahashi, and Akihiro Sugimoto. Image categorization using scene-context scale based on random forests. *IEICE Trans. Inf. Syst.*, 94-D(9):1809–1816, 2011. doi: 10.1587/transinf.E94.D.1809. URL `https://doi.org/10.1587/transinf.E94.D.1809`.

[10] Minho Kim. Fast9-accelerator, June 2017. URL `https://github.com/ISKU/FAST9-Accelerator/blob/master/C++/FAST9/fast9.cpp`.

[11] Andreas Kölsch, Muhammad Zeshan Afzal, Markus Ebbecke, and Marcus Liwicki. Real-time document image classification using deep cnn and extreme learning machines. In *2017 14th IAPR International*

Conference on Document Analysis and Recognition (ICDAR), volume 01, pages 1318–1323, 2017. doi: 10.1109/ICDAR.2017.217.

[12] David G. Lowe. Distinctive image featuresfrom scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[13] Vladimir Nekrasov, Chuunhua Shen, and Ian Reid. Light-weight refinenet forreal-time semantic segmentation. October 2018.

[14] Edward Rosten, Reid Porter, and Tom Drummond. Faster and better: A machine learning approach to corner detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(1):105–119, 2010. doi: 10.1109/TPAMI.2008.275.

[15] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International Conference on Computer Vision*, pages 2564–2571, 2011. doi: 10.1109/ICCV.2011.6126544.

[16] Jie Zhu and Zhiqian Chen. Real time face detection system using adaboost and haar-like features. In *2015 2nd International Conference on Information Science and Control Engineering*, pages 404–407, 2015. doi: 10.1109/ICISCE.2015.95.

[17] Fei Zuo and P.H.N. de With. Real-time facial feature extraction using statistical shape model and haar-wavelet based feature search. In *2004 IEEE International Conference on Multimedia and Expo (ICME) (IEEE Cat. No.04TH8763)*, volume 2, pages 1443–1446 Vol.2, 2004. doi: 10.1109/ICME.2004.1394506.