# Face Detection On The INCA$^{+}$

# General Information

This report is a master's thesis concluding the study Computer Science. The research for this thesis was conducted at Philips Research Laboratories Eindhoven, Digital Design and Test Group.

| | |
|---|---|
| Title | Face Detection on the INCA$^+$. (Master's Thesis) |
| Author | Martijn Reuvers |
| E-mail | mreuvers@science.uva.nl |
| College Card Number | 9732128 |
| Study | Computer Science |
| Specialization | Intelligent Autonomous Systems and Intelligent Sensory Information Systems |
| Supervisors | Dr. Ir. Ben Kröse and Dr. Ir. Richard Kleihorst |

University of Amsterdam
Faculty of Science

# Abstract

Embedded systems such as mobile phones, handheld computer systems, robots etc., are becoming increasingly more important in our daily lives. There are countless examples of embedded systems on which it is feasible to have a robust face detector. For example to increase the interaction between a human and a robot, a first step for the robot could be to detect the face of the human it interacts with. However the majority of the available face detection techniques are designed to operate on high performance personal computer systems and are therefore usually not suited for embedded systems.

The primary objective of our research was to find and implement a robust face detection technique, which is able to run at a reasonable speed, on an embedded system called the INCA$^+$. We implemented two of the most promising face detection techniques on the INCA$^+$: a skin color based face detector and the Viola and Jones face detector. We used a face detection experiment to determine which of these two techniques performed best. The experiment showed that the Viola and Jones detector performed best.

Next we wanted to adapt the Viola and Jones detector so that it could run on the low-level SIMD processor which is part of the INCA$^+$. To increase the performance of the Viola and Jones detector, we propose two improvements of the algorithm. As a general improvement of the Viola and Jones algorithm, the usage of multiple threshold weak classifiers is proposed. A second improvement considers the usage of weak classifiers containing complex (Gabor) filters, instead of the simple Haar-like filters used in the original algorithm. Computing these complex filters would normally be computationally too expensive. We will demonstrate that when these complex filters are separable they can be computed rapidly on an SIMD processor.

**Keywords:** Face Detection, Skin Color Based Face Detection, Viola and Jones Face Detector, SIMD Processor, Separable Filters, Gabor Filters, Multiple Threshold Weak Classifiers.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Face detection is the ability to detect and localize faces within an image or scene. Evolved through millions of years, humans can perform this task effortlessly. The human skill to detect faces in an image or scene is very robust despite various lighting conditions, scene conditions, facial occlusions or different facial features (e.g. beard, mustache, glasses etc.). Humans practically never fail in detecting faces in a scene or image.

Face detection should not be confused with face recognition, which is the ability to recognize a face given a facial image. Face detection can therefore be seen as a necessary first step in a face recognition system: first faces are detected and localized. Next the face recognition system tries to recognize the detected faces. Biological evidence indicates that this two step approach is also present in the human brain [42, 30]. People who suffer from prosopagnosia, a face recognition impairment, are not able to recognize otherwise familiar faces [10]. However they are able to perform the face detection task (i.e. they are able to detect and localize faces in images or sceneries). This further illustrates the importance of face detection.

## 1.1 Face Detection On A Computer

The ability to detect faces in a scene is critical for humans in their everyday activities. Consequently, automating this task would be useful in many application areas such as intelligent human-computer interfaces, content-based image retrieval, security, surveillance, gaze-based control, video conferencing, speech recognition assistance, video compression as well as many other areas. However it was not until recently that face detection gained a strong popularity among the computer vision community, simply because the processing power of early processors was not sufficient to perform such a difficult task.

Some of the factors that make face detection such a difficult task are [50]:

- **Face orientation.** A face can appear in many different poses. For instance the face can appear in a frontal or a profile (i.e. sideways) position. Furthermore a face can be rotated in plane (e.g. it appears under an angle of 45°). Therefore a face appears in many different shapes in an image.
- **Face size**. The size of the human face can vary a lot. Not only do different persons have different sized faces, also faces closer to the camera appear larger than faces that are far away from the camera.
- **Different facial expression.** The appearance of a person who is laughing is totally different than the appearance of a person who is angry. Therefore facial expressions directly affect the appearance of the face in the image.
- **Different facial features.** Some people wear glasses, some have a beard or a mustache, others have a scar. These type of features are called facial features. There are countless examples of facial features and they all vary in shape, size and color.
- **Occlusion.** Faces in images may be partially occluded. For instance a person standing in front of another or an object that is placed in front of the face. Therefore only part of the facial image is present in the image.
- **Lighting conditions.** Faces appear totally different when different lighting conditions are used. For instance when side lighting is used, a part of the face is very bright while the other part is very dark.
- **Scene conditions.** The scene in which the face is placed ranges from a simple uniform background to high detailed complex backgrounds. In the latter case it obviously is more difficult to detect a face.

In the past several years, many face detection techniques have been proposed throughout the literature that solve many of the above problems. However there is still no single face detection technique available that fully solves all problems.

### 1.1.1 Face Detection On Embedded Systems

Embedded systems such as mobile phones, handheld computer systems, robots, smart vacuum cleaners, autonomous surveillance video systems etc., are becoming increasingly more important in our daily lives. There are countless examples of embedded systems on which it is feasible to have a robust face detector. For example to increase the interaction between a human and a robot, a first step for the robot could be to detect the face of the human it interacts with. This could enable a robot to look at a person when the person communicates with it. Another example of face detection in an embedded environment could be video conferencing via the (mobile) telephone. Instead of sending the complete image including the background, only the region containing the face is send over the network, saving bandwidth. As the demand for embedded systems will grow, so will the demand for embedded face detection systems.

## 1.2 Problem Definition And Objectives

The majority of the available face detection techniques are designed to operate on high performance personal computer systems. Since embedded systems often require a low-cost processor (otherwise the device would be too expensive) that has a low-power consumption (due to battery constraints or to prevent overheating), the performance of these processors is usually much lower than the performance of processors in personal computers systems. Therefore many of the face detection techniques that run at high speeds on personal computers are not suited to run on an embedded system. Although some attempts have been made to implement a face detector on an embedded system [43, 26], these systems often utilize hardware specifically designed for the face detection task, making it harder to adapt them for the detection of other objects.

### 1.2.1 Research Objectives

The primary objective of our research was to find and implement a robust face detection technique, which is able to run at a reasonable speed, on an embedded system called the INCA$^+$. The INCA$^+$ is an intelligent camera equipped with two processors: a high-level processor called “TriMedia” and a low-level parallel processor called “Xetal”. The TriMedia processor is able to execute programs that are designed for personal computers. However the performance of the TriMedia is much lower than the average processor equipped in nowadays personal computer systems. The second processor is a low-cost, low-power consuming parallel processor which could easily be used in other embedded systems. More information on the INCA$^+$ and its processors is presented in Chapter 2.

The second objective of our research was to investigate the possibility to adapt the chosen face detection technique to enable it to run on the Xetal processor. If we prove it to be possible to implement a robust face detector, or at least a significant part of it, on the Xetal processor, it could be a first step in creating a robust low-cost, low-power consuming face detector which can be used in many embedded systems. Furthermore an implementation of the face detector on the low-level Xetal also reliefs the high-level TriMedia from doing the computationally expensive face detection task. This enables the TriMedia processor to spend most of its processing time on other tasks (e.g. face recognition).

Finally our third objective was to try to improve the chosen face detection technique in terms of performance and speed.

## 1.3 Thesis Overview

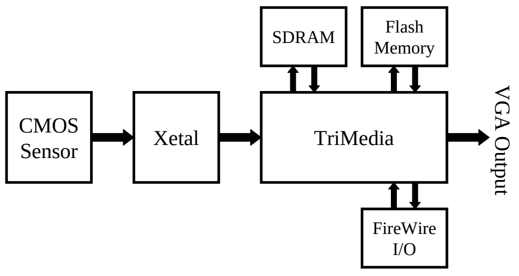
In Chapter 2 we will present the architecture of the INCA$^+$ in more detail. Subsequently, a modest literature survey is presented in Chapter 3, which outlines several face detection techniques. Based on this survey we choose two of the most promising face detection techniques that suit best on the INCA$^+$. Chapter 4 describes an experiment which we used to select the best of the two face detection techniques. In Chapter 5 we discuss the possibility to adapt the chosen face detection technique enabling it to run on Xetal. Chapter 6 will present some possible improvements on the chosen face detector. Finally, Chapter 7 concludes this thesis with a summary of the work and the direction of future research and improvements.

# Chapter 2

# Architecture Of The INCA$^+$

As already mentioned in the introduction, the face detector has to run on a platform called the INCA$^+$. The INCA$^+$ is an intelligent camera produced by Philips CFT and is shown in Figure 2.1(a). This camera houses a CMOS sensor, a parallel processor called “Xetal” for low level pixel operations, a DSP called “TriMedia” for the high level programs and some communication devices to communicate with for instance a PC as depicted in Figure 2.1(b). The next sections discuss the two major components of the INCA$^+$: Xetal and TriMedia. First a brief introduction on SIMD architectures is presented in order to better understand the Xetal architecture.

(a) Outside



(b) Inside

**Figure 2.1:** The INCA$^+$

## 2.1 Introduction To SIMD

Xetal is a parallel chip working in SIMD mode. SIMD stands for Single Instruction, Multiple Data. An SIMD processor contains many processing elements (PEs) combined into a Linear Processor Array (LPA). Each PE performs the same operation only on different data. A typical PE usually contains an arithmetic unit (AU) and a memory unit (MU). The program flow is controlled by the Global Control Processor (GCP) which controls the program counter and passes the instructions to every PE in the LPA. To illustrate the working of an SIMD processor in Figure 2.2 an example of a single increment (`INCR`) instruction on SIMD is shown.

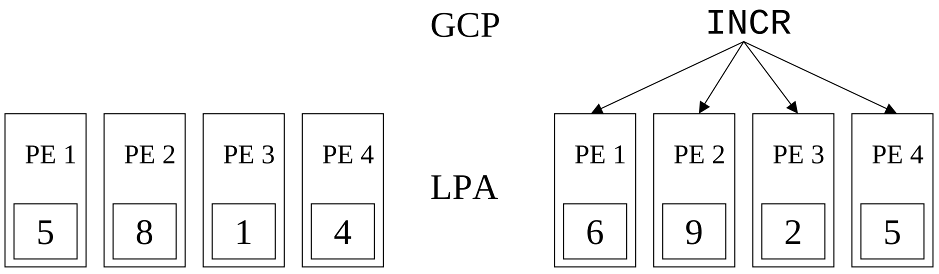


**Figure 2.2:** Example of the increment operation on SIMD. There are four PEs in the LPA, each operating on its own memory element (the rectangles within the PE). The GCP passes one `INCR` instruction to the four PEs and the contents of all four memory elements are incremented in a single instruction. On a sequential processor this would require four instructions, since each element would have to be incremented separately.

### 2.1.1 Memory Models

There are roughly two memory models for an SIMD processor [5]: a distributed and a shared memory model. In case of an SIMD processor using a distributed memory model each PE is only allowed to use its own memory. If a PE needs the information contained in another PE, it has to put in a request to the GCP and the GCP manages the transferring of information. The advantage of this memory model is the ease of adding more memory and PEs to the processor. The disadvantage of the distributed model can be found in the time wasted by the GCP managing all memory exchanges.

In case of a shared memory model each PE shares its memory with other PEs. The memory elements of each PE are connected through a network or a switching unit. This reliefs the GCP from doing costly memory exchanges. The disadvantage of this model is inherited in the difficulty of adding memory. In this thesis only shared memory models are considered. The shared memory model can further be subdivided into fully and limited connected memory models[1]:

- In case of a fully connected shared memory model, each PE is connected to every other PE. The advantage of fully connected models is that each PE can read directly from every other PE. The disadvantage is the fact that the number of wires required to make all connections grows exponentially with the number of PEs. For this reason fully connected models are hardly ever used when there is a large number of PEs in the LPA.

- In case of a limited connected shared memory model, each PE is connected to $n$ other PEs. The advantage of this model is that only a limited number of wires is needed per PE for the connections. The disadvantage of limited connected models is that each PE can only read directly from its $n$ connected PEs. Shifting is used to reach other PEs as shown in Figure 2.3. A problem with shifting is that the number of shifts increases linearly with the distance between the source and destination PE. Fetching values from far away PEs therefore consumes a lot of instructions.

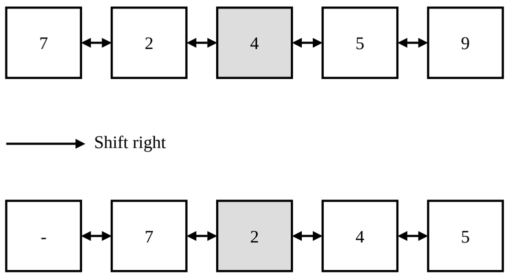


**Figure 2.3:** Shift example. In this figure five PEs are drawn and each PE is connected only to its left and right neighbor. Suppose that the center PE wants to read the value 7 of the leftmost PE. But the center PE can only read the value 2 and 5 from its direct neighbors. In order to reach value 7 each PE reads and stores the value from its left neighbor, effectively shifting the whole row to the right. Now the center PE can read value 7. Note that in a fully connected memory model value 7 could be fetched immediately.

[1] Other interconnection topologies such as rings, stars, trees and hypercubes are also possible but they are beyond the scope of this thesis.

## 2.2 The Xetal Architecture

Xetal's LPA contains 320 PEs, each with its own AU and MU. The AU contains an accumulator, an adder and a multiplier with which comparison, addition, subtraction, data weighing and multiply-accumulate can be performed [4, 20]. The PEs also incorporate a flag that is used for conditional pass-instructions. Xetal uses the limited connected shared memory model as described in the previous section. In Figure 2.4 a detailed overview of

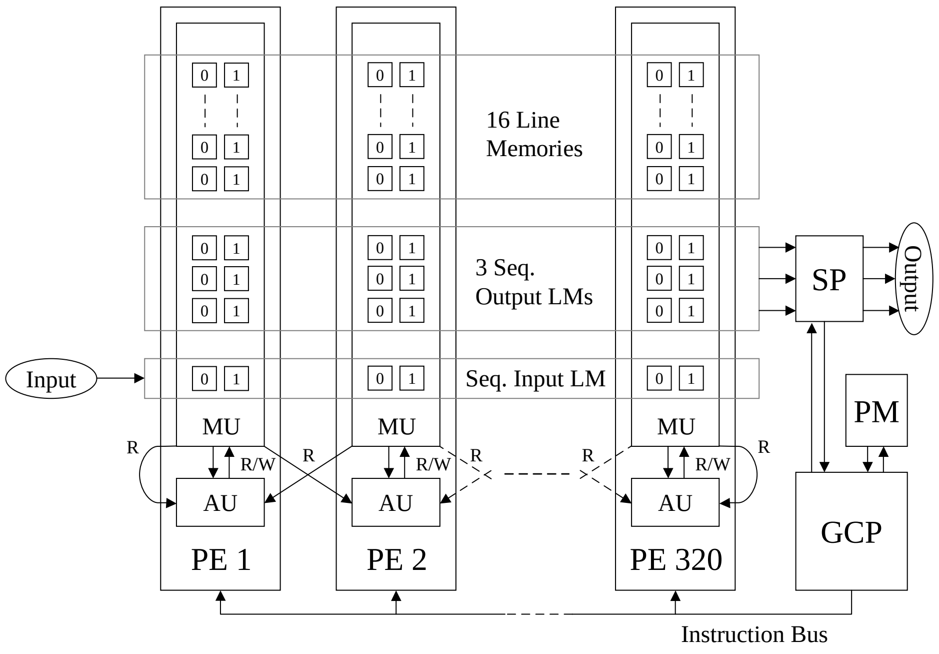


**Figure 2.4:** Detailed overview of the Xetal processor

Xetal is presented. As shown the AU operates on two memory columns because the image is 640 pixels in width and there are 320 processing elements (PEs). Each memory element is 10 bits in size. A single frame contains 480 lines and the AU can execute a maximum of 1560 instructions per PE per line. Each AU is also connected to the MU of its direct neighbors, making it possible to read from them. Since there is no left neighbor of PE 1 and no right neighbor of PE 320, reading from that neighbor results in reading from their own MU.

The image or video input data is a VGA (640$\times$480 pixels) size frame with up to 10-bit digitized signals at a maximum rate of 30 frames/second. This data is read per line into the sequential input line memory. The raw data originating from a sensor, can then be converted to any popular format (e.g. RGB or YCbCr) using interpolation and some line memories for temporary storage. The converted format is sent out to the serial processor via three sequential output line memories. The serial processor (SP) reads out the three sequential output line memories and monitors the statistics (minimum value, maximum value and average value) of the image data, which can be read by the GCP. The data is then sent off-chip using three ports, each 10-bit wide. The Program Memory (PM) contains the actual program code. The GCP reads from the PM and sends the instructions to each PE via the Instruction Bus. Details on how to program Xetal as well as a number of programming examples can be found in Section B.1.

## 2.3 The TriMedia Processor

The second processor in the INCA$^+$ is the TriMedia 1300 32-bit processor running at 133 MHz. This processor uses a Very Long Instruction Word (VLIW) architecture which means that every instruction can execute up to five operations in parallel in a single cycle, as shown in Figure 2.5. Each operation can again contain multiple arithmetic operations, for example the `ifir(a,b)` operation contains a total of three arithmetic operations: two multiplications and one addition ($\texttt{aHI} \times \texttt{bHI} + \texttt{aLO} \times \texttt{bLO}$) [3]. However the performance of the operation parallelism is entirely dependent on the performance of the compiler. In the best case five operations can run in parallel. But in the worst case no operations run in parallel and only one operation is executed per instruction.

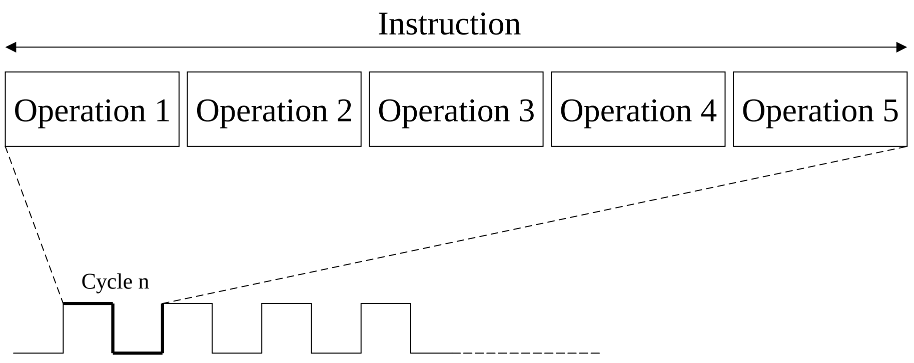


**Figure 2.5:** A VLIW instruction

Furthermore, the TriMedia 1300 contains an image coprocessor, which can be used for image processing tasks (e.g. image scaling), and a floating point coprocessor (FPU) which reliefs the processor from doing costly floating point operations. Details on how to program TriMedia can be found in Section B.2.

## 2.4 Architecture Discussion

This chapter showed that Xetal is suited for doing low-level operations. Algorithms that are not complex of nature but do require a lot of processing time suit best on Xetal. Xetal's architecture is simple and it has a low power consumption, allowing it to run at a relatively high performance as compared to other parallel architectures (e.g. MIMD, which requires a lot more power to achieve the same performance). Also a low power consumption lowers the risk of overheating, which is especially useful in an embedded architecture.

The TriMedia on the other hand is more suited for doing more complex higher level algorithms. Its architecture is designed to work with large and complex datastructures and has a more extended instruction set as compared to Xetal.

The ideal setup for any application on the INCA$^+$ would be to (pre) process as much as possible on Xetal. Further (post) processing can then be done by the TriMedia. For instance the ideal face detection application would have a large part running on Xetal. This reliefs the TriMedia from doing computationally expensive low level operations, enabling it to spend more time on more sophisticated applications (i.e. face recognition using a database).

In the following chapter we will discuss various face detection techniques presented throughout the literature. We will determine which of these techniques could be implemented on the INCA$^+$ architecture.

Chapter 3

# Face Detection: A Literature Survey

We can roughly group the existing face detection techniques into two categories: techniques that are *feature-based* and techniques that make use of *sliding windows*.

Feature-based techniques follow a bottom-up approach, first (low level) features are extracted from an input image (i.e. image segmentation). Second a high level knowledge based system analyses the features and determines whether the features belong to a face or not. Feature-based techniques usually operate on pixel level (e.g. skin color), or use a small filter kernel (e.g. edge-based techniques). Feature extraction usually is a simple low level task which can be done with little effort. For instance a check to determine whether a pixel is a skin color pixel or not only takes a few instructions. This simplicity allows the feature extraction to run at high speeds. A drawback of feature-based techniques is that they often operate on pixel level and therefore often suffer from ambiguity due to illumination, signal noise and occlusion [50, 16].

Sliding window techniques follow a top-down approach. These techniques use a detection window of a fixed size and place this detection window over the input image. Next the algorithm determines whether the content of the image inside the window represents a face or not. When it finishes processing the window content, the window "slides" to another location on the input image and the algorithm again tries to determine whether the content of the detection window represents a face or not. This procedure continues until all image locations are checked. To detect faces at different scales usually the image is scaled down a number of times to form a so-called "image pyramid" (see Figure 3.5) and the detection procedure is repeated. Sliding window based techniques are usually more robust, however they also require a lot more processing time.

The following sections will discuss a number of face detection techniques. First skin color based face detection, a very popular feature-based technique, will be reviewed. Subsequently we will discuss four sliding window techniques. This chapter concludes with a discussion on which of these techniques fits best on our architecture.

## 3.1 Skin Color Based

Skin color based face detection methods have gained strong popularity among the face detection community, mainly because of its simplicity and robustness to geometric variations of the face. Furthermore the skin color of humans of different races, although perceived differently by humans, only differs in intensity rather than chrominance. This chrominance invariance of the human skin makes it possible to implement a simple and consistent skin color segmentation method. Skin color based face detection techniques can be divided into three steps:

- Colorspace decision.
- Skin model creation and segmentation of the image using that model.
- Face localization using the segmented image.

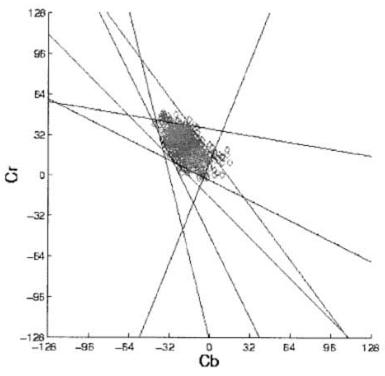
For skin based face detection many colorspaces have been proposed throughout the literature. Some popular examples of colorspaces are: RGB, Normalized RGB, YCbCr (i.e. YUV), HSI (Hue, Saturation and Intensity) as well as many others. Which colorspace is the best for skin color segmentation is still subject to discussion. In [37] Shin *et. al.* showed that the separability between skin and non-skin colors depends on the chosen colormodel. Using four separability metrics on 18 different colorspaces, they concluded that the RGB and YCbCr colorspace performed best. However these conclusions were later questioned by Vezhnevets *et. al.* in [45].

The next step is to create a skin color model, based on the chosen colorspace. This model is used to separate the skin pixels from the non-skin pixels (i.e. segmentation). There are many proposed skin modeling methods, ranging from simple segmentation rules [19, 15] to more complex statistical models [49] and adaptive methods (e.g. neural networks [8]).

A problem surfaces when different illumination conditions are used. The color of objects and scenes change dramatically when placed in a different lighting condition[1]. Several approaches have been proposed to address this problem, however they require temporal knowledge [27] or require the camera characteristics and light source spectrum to be known [39].

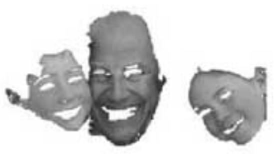
The last step involves a high level knowledge-based system that tries to determine the location of the faces by using the segmented image. Many

[1] This is a general problem in color vision and is called *color constancy*.

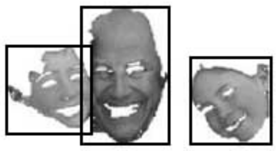


(a) Choose segmentation rules

(b) Original image



(c) Skin segmentation



(d) Localizing the faces

**Figure 3.1:** A typical skin color based face detection approach. After choosing the desired colorspace the skin segmentation rules are chosen (a). Next the original image (b) is segmented based on these rules (c). In the last step a high level system analyzes the segmented image and localizes the faces.

approaches have been proposed throughout the literature. We will address some of these approaches. In [38] Singh *et. al.* extract facial features such as the eyes and the mouth from the image, based on the assumption that eyes and mouth are regions that appear darker in an image. These facial features are used for face verification. In [47] Wang *et. al.* first apply a contour finding algorithm on the binary skin map. Next they try to find facial features within the found contour and try to determine the symmetry axis of the face based on the position of the mouth. In [35] Saber *et. al.* construct an elliptical model of the face and try to match the shape of a binary skin region to that elliptical model (i.e. ellipse fit). Once the elliptical facial pattern is found, they try to find facial features within this pattern.

### 3.1.1 Discussion

As with all feature-based methods, the performance of the skin color based face detector heavily depends on the performance of the segmentation method. If the performance is bad, the high level system has a very difficult task in trying to locate the human face in the segmented image. Many of the reported results were obtained using face images that were taken in conditioned environments (e.g. frontal non-colored lighting). The reported detection rate in these environments is usually very high (i.e. detection rate between 80% and 95% [38, 15]).

## 3.2 Distribution-Based

In [40] Sung and Poggio developed a distribution-based method for face detection. Their system, which is a sliding window based technique, first models $19 \times 19$ sample images into multi-dimensional sample image vectors. Next they try to subdivide the sample space, containing the sample image vectors, into subclasses. To approximate the subclasses they use multi-dimensional Gaussian clusters. The system developed by Sung and Poggio consists of the following four steps ([16, 50, 40]):

1. First the image in the detection window is preprocessed by rescaling it to $19 \times 19$ pixels and applying the steps shown in Figure 3.2. This preprocessing step enhances the image and reduces the dimensionality of the image vector from $\Re^{361}$ to $\Re^{283}$.

2. A distribution-model of canonical face- and nonface patterns is constructed using 12 multi-dimensional Gaussian clusters as shown in Figure 3.3. The 283-dimensional clusters are constructed using a modified $k$-means clustering algorithm which computes the cluster centroids and covariance matrices.

3. Given a new image, the distance between that image pattern and each cluster is computed, resulting in 12 distances between the image and the 12 cluster centroids. For each of the 12 distances, two values are computed. The first value is the Mahalanobis-like distance between the new image and the cluster centroid in a subspace spanned by the cluster's 75 largest eigenvectors. The second value is the Euclidean distance between the new image and its projection in the subspace. Given 12 distances and two values per distance, in total a 24-dimensional image measurement vector is obtained.

4. A multilayer perceptron (MLP) is used to classify input patterns as faces or nonfaces, using the 24-dimensional image measurement vector. The MLP is trained using standard backpropagation from a training set of 47 316 patterns of which 4150 are face patterns.

To classify a new input pattern, the image pattern is preprocessed (step 1) and subsequently the 24-dimensional image measurement vector is computed (step 3). Finally the MLP determines whether the image measurement vector corresponds to a face image or not.

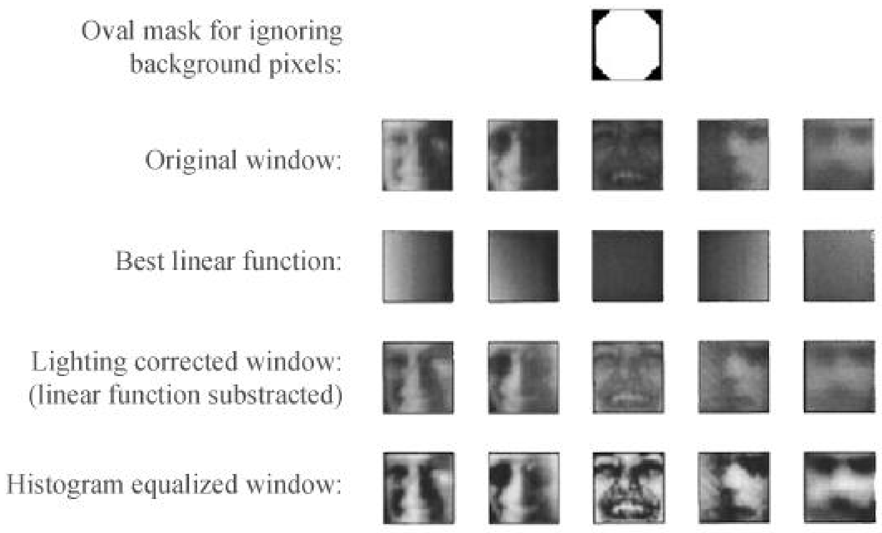

Oval mask for ignoring background pixels:
Original window:
Best linear function:
Lighting corrected window: (linear function substracted)
Histogram equalized window:

**Figure 3.2:** The preprocessing step developed by Sung and Poggio [40]. It preprocesses the image by first applying an oval mask for eliminating near-boundary pixels from the image. This step also reduces the dimension of the image vector from $\Re^{361}$ to $\Re^{283}$. It continues by searching for a best fit brightness plane in the masked image pixels and subsequently it subtracts that brightness plane from the unmasked image pixels. Finally histogram equalization is applied to correct for different camera gains and to improve the contrast of the image.

### 3.2.1 Bootstrapping

Because the negative examples are difficult to characterize and the negative nonface class is far more broader and richer than the positive face class, more negative examples are needed to accurately separate the negative from the positive class. To do this Sung and Poggio used a so called “bootstrapping” method. This bootstrapping method uses images that do not contain any faces. During each training session misclassifications (false positives) on these images are placed in the training set for the next session. This overcomes the problem of using a huge set of nonface images in the training set, many of which may not influence the training. Good sources of false positives are images of landscapes, buildings, trees etc., due to the different textured patterns they contain [28].

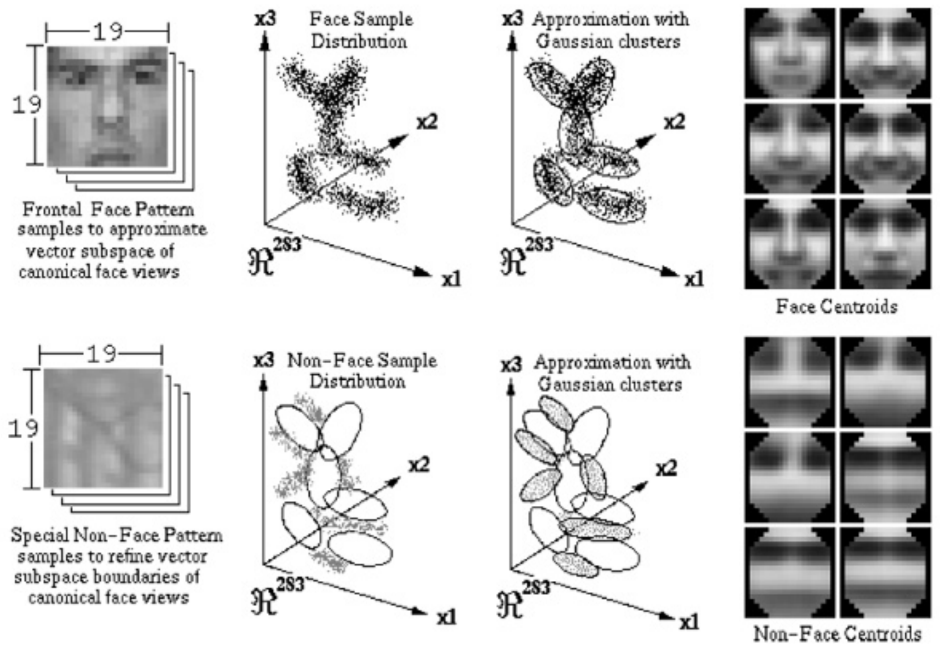

19
19
Frontal Face Pattern samples to approximate vector subspace of canonical face views
x3
Face Sample Distribution
x2
$\Re^{283}$
x1
Approximation with Gaussian clusters
Face Centroids
Special Non-Face Pattern samples to refine vector subspace boundaries of canonical face views
Non-Face Sample Distribution
Non-Face Centroids

**Figure 3.3:** The distribution based face model as proposed by Sung and Poggio [40]. The top row shows an empirical distribution of face patterns. They model this distribution using six multi-dimensional Gaussian clusters, whose centers are shown on the right. The bottom row shows the distribution of nonface patterns and again they model this by using six multi-dimensional Gaussian clusters. In total their model consists of 12, 283-dimensional clusters.

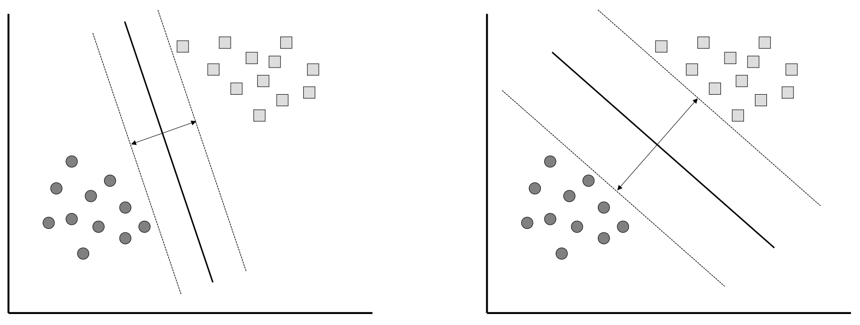
### 3.2.2 Results

Sung and Poggio tested their system on two different facial databases. They reported a detection rate of 96.3% with three false detections on the first database containing 301 frontal and near-frontal face mugshots of 71 different persons. On their second test set (the MIT subset of the MIT+CMU facial database [41]) they reported a detection rate of 79.9% with 5 false positives. However they did not report the speed of their detector.

## 3.3 Support Vector Machines

Support Vector Machine (SVM) is a pattern classification algorithm developed by Vapnik *et. al.* [44]. Most machine learning based classification techniques are based on the principle of minimizing the error in training data, called empirical risk minimization. SVMs operate on another induction principle, called structural risk minimization, which minimizes an upper bound on the generalization error.

For classification, SVMs operate by finding a hyperplane in the space of possible inputs. This hyperplane will attempt to split the positive examples from the negative examples. The hyperplane will be chosen to have the maximum distance to the nearest of the positive and negative examples as depicted in Figure 3.4. Intuitively, this makes the classification correct for testing data that is near, but not identical to the training data.



(a) Small distance. (b) Maximum distance

**Figure 3.4:** Separating hyperplane in the SVM [44]. In (a) a hyperplane with a small distance to the nearest of the positive and negative examples is chosen, yielding worse generalization. In (b) the hyperplane has a maximum distance between the two classes, hence a better generalization capability is expected.

Support Vector Machines were first used for face detection by Osuna *et. al.* in [28]. They trained the SVM using a database of faces and non-faces of size 19 × 19 pixels. The proposed system uses a preprocessing step adopted from Sung and Poggio's system [40] which is shown in Figure 3.2. This preprocessing step reduces the dimensionality of the input space and improves the image quality. Osuna *et. al.* also use the bootstrapping method as explained in the previous section.

### 3.3.1 Results

Osuna *et. al.* tested their SVM on two facial databases. The first set contained 313 high-quality images with a single face per image. The second set contained 23 images of mixed quality and a total of 155 faces. For comparison reasons they also tested the performance of Sung and Poggio system on these databases. In total 4 669 960 pattern windows were evaluated for the first database and 5 383 682 for the second. The detection rate and the number of false positives are presented in Table 3.1. The SVM performed better when looking at the detection rates (DR), and slightly worse when looking at the number of false positives (FP). However Osuna *et. al.* claimed that their system was 30 times faster than the system of Sung and Poggio.

| | Database 1 | | Database 2 | |
|---|---|---|---|---|
| | DR | FP | DR | FP |
| SVM | 97.1% | 4 | 74.2% | 20 |
| Sung *et. al.* | 94.6% | 2 | 74.2% | 11 |

**Table 3.1:** Performance of the SVM face detector versus the face detector of Sung and Poggio.

## 3.4 Neural Networks

Since face detection can be considered as a two-class pattern recognition problem, various neural network architectures have been proposed throughout the literature. Probably the most significant work on face detection using neural networks has been done by Rowley *et. al.* [33]. Their system basically consists of three steps: preprocessing, the neural network and postprocessing.

Rowley *et. al.* also adopt the preprocessing method devised by Sung and Poggio [40] shown in Figure 3.2. The preprocessed images are passed to the neural network. The neural network is designed to process images of $20 \times 20$ pixels, yielding 400 input units. There is one hidden layer with 26 units: 4 units process $10 \times 10$ pixel subregions, 16 units process $5 \times 5$ subregions and 6 process $20 \times 5$ horizontal stripes as shown in Figure 3.5.

The network was trained using 1050 face samples of various sizes, intensities, orientations and positions. 1000 nonfacial images were used for training and were taken randomly from a set of images not containing any faces. Given a test pattern, the trained neural network outputs a value ranging from -1 (nonface) to 1 (face).

The postprocessing step counts the number of detections in a small neighborhood and if it is above a certain threshold a face is likely to be present. The postprocessing step also merges overlapping detection rectangles.To further improve performance, Rowley *et. al.* trained multiple neural networks and combined the output with simple arbitration schemes such as logic operators (AND/OR) and voting.

A limitation of the system is that it is only able to detect upright, frontal faces. Rowley *et. al.* extended their method in [34] to detect faces at all angles in the image plane.

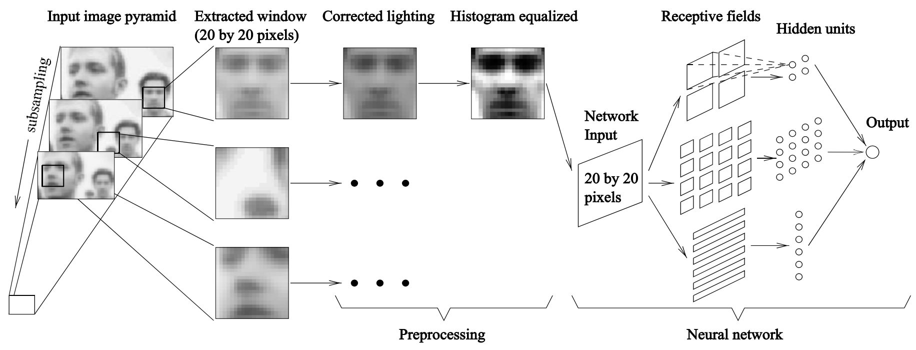


**Figure 3.5:** The system of Rowley *et. al.* [33]

### 3.4.1 Results

For upright faces Rowley *et. al.* claim a detection rate between 77.9% and 90.3% of faces in a set of 130 test images, with an acceptable number of false detections. Depending on the application, the system can be made more or less accurate by varying the arbitration heuristics or thresholds used.

The system has been tested on a wide variety of images, with many faces and unconstrained backgrounds. A fast version of the system can process a $320 \times 240$ pixel image in 2 to 4 seconds on a 200 MHz R4400 SGI Indigo 2.

## 3.5 Viola And Jones Face Detector

In [46] Paul Viola and Michael Jones describe a novel object detection method which can be used for face detection. Their sliding window based algorithm uses so-called filters[2] rather than operating on raw pixels, like for instance Rowley *et. al.* do in their neural network approach. The filters span a rectangular area of pixels within the detection window, and can be computed rapidly when using a new image representation called the integral image.

The number of possible filters within a detection window is very large. In order to select only the most effective filters, the object detector is trained by using an adapted version of the AdaBoost algorithm, originally proposed by Freund and Schapire in [13]. This boosting algorithm selects a small number of so-called weak classifiers (each containing a single filter) from a large pool of weak classifiers. None of these weak classifiers will have a high

[2] Instead of the word "filter", Viola and Jones used a different term: "feature". However as we already used that word for feature-based techniques we decided to use the word "filter" to avoid ambiguity. The word "filter" instead of "feature" for this purpose is also used in [36].

classification performance, however when they are combined they form a strong classifier which is able to achieve a high classification performance.

To further reduce the number of weak classifier evaluations per detection window, Viola and Jones propose the use of an "attentional cascade". This cascade combines successively more complex classifiers into a cascade structure. Using this layered approach with increased complexity it is possible to reject non-face windows very quickly while spending more computational effort on windows that are more likely to contain a face.

This section will discuss the Viola and Jones method more in-depth since it plays an important role in this thesis. First we will discuss the filters used by Viola and Jones. Then we will describe the integral image and how these filters can be computed using the integral image. Subsequently we will describe the boosting algorithm used to reduce the number of weak classifiers to form an effective strong classifier. Next the attentional cascade and the experimental results obtained by Viola and Jones is discussed. We will conclude this section with a brief overview on related work.

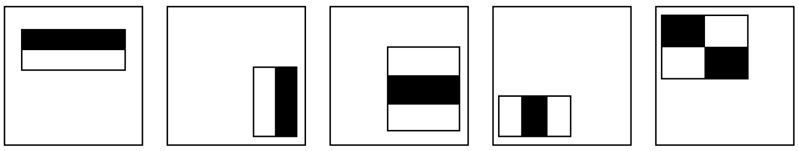
### 3.5.1 Filters

Viola and Jones motivated their choice for using filters rather than pixels with the following arguments: the used filters operate on a broader spatial domain enabling them to 'look' at larger structures such as edges, corners and lines inside the detection window. Second, the filters can be computed very fast when using the special image representation called the integral image. Viola and Jones used the five simple binary filters shown in Figure 3.6. These filters are partially based on work of Papageorgiou in [29], who used Haar basis functions for the detection of faces and pedestrians in images. The filters contain either two, three or four rectangular regions. The regions are equal in size and shape and are horizontally or vertically adjacent.

When using a detection window with a base size of $24 \times 24$ pixels, there are more than 180 000 different rectangular filters possible inside a single detection window. Computing all these filters per detection window would be too expensive. Therefore a small subset of these filters need to be selected to form an effective classifier. This will be discussed in Section 3.5.5.

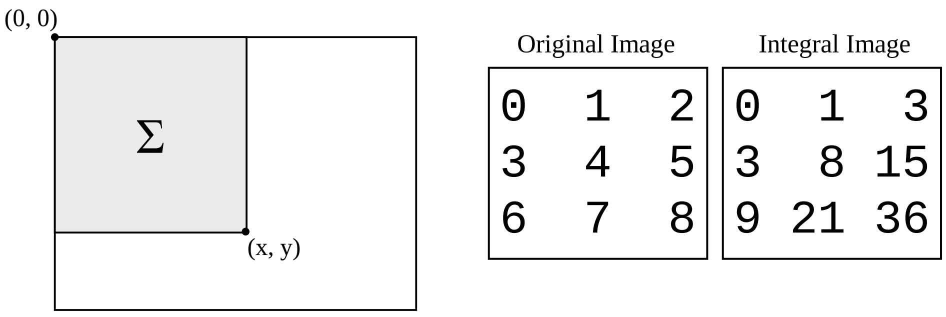
### 3.5.2 Integral Image

Viola and Jones propose a special image representation called the "integral image" to compute the rectangular filters very rapidly. The integral image is in fact equivalent to the Summed Area Table (SAT) that is used as a texture mapping technique, first presented by Crow in [9]. Viola and Jones renamed the SAT to integral image to distinguish between the purpose of use: texture mapping versus image analysis.



(a) $F_{\text{h\_edge}}$ (b) $F_{\text{v\_edge}}$ (c) $F_{\text{h\_line}}$ (d) $F_{\text{v\_line}}$ (e) $F_{\text{diag}}$

**Figure 3.6:** The five Haar like filters used by Viola and Jones placed in the detection window. To compute a filter, the sum of the pixels in the dark area is substracted from the sum of the pixels in the light area. Notice that we could also substract the light area from the dark area, the only difference is the sign of the result. The horizontal and vertical two-rectangle filters $F_{\text{h\_edge}}$ and $F_{\text{v\_edge}}$ are shown in (a) respectively (b), and tend to focus on edges. The horizontal and vertical three-rectangle filters $F_{\text{h\_line}}$ and $F_{\text{v\_line}}$ are shown in (c) respectively (d), and tend to focus on lines. The four-rectangle filter $F_{\text{diag}}$ in (e) tends to focus on diagonal lines [24].



Original Image

| | | |
|---|---|---|
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Integral Image

| | | |
|---|---|---|
| 0 | 1 | 3 |
| 3 | 8 | 15 |
| 9 | 21 | 36 |

(a) Integral image value at point $(x, y)$ (b) Integral image value in numbers

**Figure 3.7:** The integral image value.

The integral image at location $(x, y)$ is defined as the sum of all pixels above and to the left of $(x, y)$ (see Figure 3.7(a)):

$$ii(x, y) = \sum_{j=0}^{x} \sum_{k=0}^{y} I(j, k) \tag{3.1}$$

where $ii(x, y)$ is the integral image value at $(x, y)$, and $I(x, y)$ is the original image value. This equation can be rewritten into the following recurrent equations:

$$r(x, y) = r(x, y - 1) + I(x, y) \tag{3.2}$$

$$ii(x, y) = ii(x - 1, y) + r(x, y) \tag{3.3}$$

where $r(x, y)$ is called the cumulative row sum[3], $r(x, -1) = 0$, $ii(-1, y) = 0$ and $ii(x, -1) = 0$. Using these recurrent equations the integral image can be computed in a single pass over the original image.

### Computing A Rectangular Pixelsum

The pixelsum of a rectangular area is defined as:

$$\text{pixelsum}(x, y, w, h) = \sum_{j=x}^{x+w-1} \sum_{k=y}^{y+h-1} I(j, k)$$

where $w$ and $h$ are the width and height respectively of a rectangular area at location $(x, y)$ and $I(j, k)$ is the image value at location $(i, j)$. We will now demonstrate that this value can be computed rapidly when using the integral image. The integral image value at the lower right corner of a rectangular area is defined as:

$$ii(x + w - 1, y + h - 1) = \sum_{j=0}^{x+w-1} \sum_{k=0}^{y+h-1} I(j, k)$$

We can rewrite this into the following four terms:

$$\begin{aligned} ii(x + w - 1, y + h - 1) \quad &= \sum_{j=0}^{x-1} \sum_{k=0}^{y-1} I(j, k) + \sum_{j=0}^{x-1} \sum_{k=y}^{y+h-1} I(j, k) + \\ &\sum_{j=x}^{x+w-1} \sum_{k=0}^{y-1} I(j, k) + \underbrace{\sum_{j=x}^{x+w-1} \sum_{k=y}^{y+h-1} I(j, k)}_{\text{pixelsum}} \end{aligned}$$

When looking at these four terms it can be observed that the last term equals the pixelsum. To obtain this value we need to substract the other three terms. We can do this when using three other integral images values (see Figure 3.8):

$$\begin{aligned} ii(x - 1, y - 1) \quad &= \quad \sum_{j=0}^{x-1} \sum_{k=0}^{y-1} I(j, k) \\ ii(x + w - 1, y - 1) \quad &= \quad \sum_{j=0}^{x+w-1} \sum_{k=0}^{y-1} I(j, k) \\ ii(x - 1, y + h - 1) \quad &= \quad \sum_{j=0}^{x-1} \sum_{k=0}^{y+h-1} I(j, k) \end{aligned}$$

[3] The term cumulative row sum was introduced by Viola and Jones but is rather ambiguous. Unlike its name suggests it is a column containing the summed $y$-values. Thus the cumulative row sum at position $(x, y)$ contains the sum of all $y$-values of the rows above $(x, y)$: $r(x, y) = \sum_{k=0}^{y} I(x, k)$.

These values can be rewritten into their following equivalents:

$$\begin{aligned}
ii(x-1, y-1) &= \sum_{j=0}^{x-1}\sum_{k=0}^{y-1} I(j,k) \\
ii(x+w-1, y-1) &= ii(x-1,y-1) + \sum_{j=x-1}^{x+w-1}\sum_{k=0}^{y-1} I(j,k) \\
ii(x-1, y+h-1) &= ii(x-1,y-1) + \sum_{j=0}^{x-1}\sum_{k=y}^{y+h-1} I(j,k)
\end{aligned}$$

Using these three equations it is easily seen that:

$$\begin{aligned}
\text{pixelsum}(x,y,w,h) &= ii(x+w-1, y+h-1) + ii(x-1,y-1) - \\
&\quad ii(x-1,y+h-1) - ii(x+w-1,y-1)
\end{aligned}$$

Therefore when using an integral image, any rectangular pixelsum can be computed with only four lookups, two subtractions and one addition.

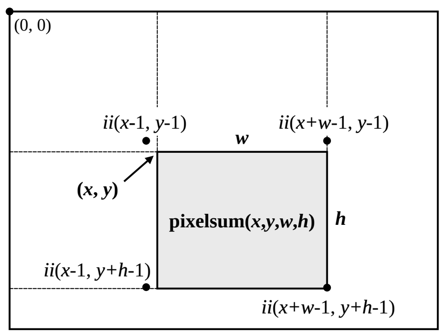


**Figure 3.8:** Calculation of the pixelsum using the integral image. Using four integral image values, $ii(x+w-1, y+h-1)$, $ii(x-1, y-1)$, $ii(x-1, y+h-1)$ and $ii(x+w-1, y-1)$ we can calculate the sum of the pixels in the grey rectangular area.

### 3.5.3 Filter Computation

The five filters as proposed by Viola and Jones consists of two or more rectangular regions which need to be added together or substracted from eachother. One of the five filters, the horizontal edge filter $F_{\text{h\_edge}}$, is shown in Figure 3.9. To compute the result $H_{\text{h\_edge}}$ of filter $F_{\text{h\_edge}}$ applied on image $I$, the sum of the pixels in the lower dark rectangular region have to be substracted from the sum of the pixels of the upper light rectangular region:

$$H_{\text{h\_edge}}(x, y) = \sum_{j=x}^{x+w-1} \sum_{k=y+\frac{1}{2}h}^{y+h-1} I(j, k) - \sum_{j=x}^{x+w-1} \sum_{k=y}^{y+\frac{1}{2}h-1} I(j, k)$$

where $w$ and $h$ are the width respectively height of the filter. When using

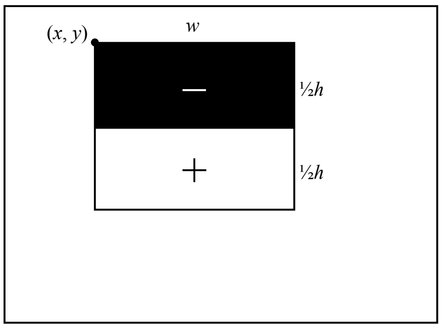
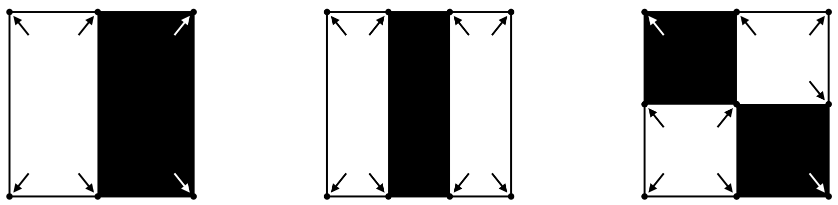


**Figure 3.9:** Computing the filter result $H_{\text{h\_edge}}$.

the integral image we can use the pixelsum as defined in Equation 3.4 and rewrite the filter result $H$ to the following:

$$\begin{aligned} H_{\text{h\_edge}}(x, y) &= \text{pixelsum}(x, y + \frac{1}{2}h, w, \frac{1}{2}h) - \\ &\quad \text{pixelsum}(x, y, w, \frac{1}{2}h) \end{aligned}$$

Using two pixelsums the filter result can be computed. Since this filter contains two adjacent rectangular areas, there are two equal integral image positions and only six integral image lookups are required to compute the filter result. The number of lookups for the other filter types is shown in Figure 3.10. Using this technique filters results can be computed very fast and in constant time regardless of their size and location. Instead of down scaling the image a number of times, Viola and Jones propose to scale up the detection window. A detection window contains only one simple filter which can be easily scaled up using extrapolation of the size and position



(a) 2 rectangles: 6 lookups (b) 3 rectangles: 8 lookups (c) 4 rectangles: 9 lookups

**Figure 3.10:** Number of lookups per filter type. A filter with two rectangular areas contains two equal integral image positions and can therefore be calculated with six lookups. A filter with three rectangular areas involve four identical integral image positions and can be calculated with eight lookups, a four rectangular filter with nine lookups.

of the filter. By scaling up the detection window there is no need to use a pyramid anymore and this saves significant computation time.

### 3.5.4 Image Normalization

Viola and Jones normalized the images to unit variance during training to minimize the effect of different lighting conditions. To normalize the image during detection they post multiply the filter results by the standard deviation $\sigma$ of the image in the detector window, rather than operating directly on the pixel values. Viola and Jones compute the standard deviation of the pixel values in the detection window by using the following equation:

$$\sigma = \sqrt{\mu^2 - \frac{1}{N}\sum x^2} \tag{3.4}$$

where $\mu$ is the mean, $x$ is the pixel value and $N$ is the total number of pixels inside the detection window. The mean can be computed by using the integral image. To calculate $\sum x^2$ Viola and Jones use a squared integral image, which is an integral image only with squared image values. Computing $\sigma$ only requires eight lookups and a few instructions.

### 3.5.5 Filter Selection Using AdaBoost

Recall that there over 180 000 filters possible within a single detection window. Computing all these filters would be too expensive. Therefore a small subset of these filters need to be selected to form an effective classifier. For this task Viola and Jones used a boosting algorithm. The original boosting algorithm called Discrete AdaBoost (Adaptive Boosting) was proposed by Freund and Schapire in [13]. They discovered an algorithm that sequentially fits so-called weak (i.e. simple) classifiers to different weighings of the samples in a dataset. The classifiers are called weak because even the best classifier will not be able to classify the training data well. Each boosting cycle the weak classifier with the lowest error is selected from a pool of weak classifiers. Next the samples that are misclassified by the weak classifier receive a greater weight on the next boosting round. The algorithm iterates several times, piling more weight on the samples that are difficult to classify. The final classifier, called a strong classifier, is a weighted combination of the weak classifiers.

Viola and Jones adapted the Discrete AdaBoost algorithm by constraining the weak classifier to use a single filter which best separates the positive and negative examples. For each filter an optimal threshold value is determined such that the minimum number of examples are misclassified. The weak classifier used by Viola and Jones is defined as:

$$h_j(x) = \begin{cases} 1 & \text{if } p_j f_j(x) < p_j \theta_j \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

where $f_j(x)$ is the result of the filter on input image $x$, $\theta_j$ is the threshold, $p_j$ is the parity which indicates the direction of the equality sign and $x$ is the input image. The learning algorithm is shown in Table 3.2.

### 3.5.6 The Attentional Cascade

There is a very large amount of possible detection windows in a single image of which the vast majority does not contain a face. Therefore detection windows not containing a face have to be rejected as fast as possible. In order to do this, Viola and Jones devised the “attentional cascade”. This cascade contains several stages (i.e. strong classifiers). The first stages in the cascade contains a small amount of weak classifiers, so these stages can be evaluated very fast. These simple stages are trained in such a way that they reject a large amount of negatives while accepting all the positives (i.e. detection rate near 100% and the false positive rate as low as possible with the fewest possible weak classifiers). Later stages of the cascade are more complex (i.e. contain more weak classifiers) and are used to lower the false positive rate. The false positive rate should be as low as possible considering the amount of possible detection windows (e.g. a $640 \times 480$ image contains over 800 000 possible detection windows when using a detection window with a base resolution of $24 \times 24$ pixels and 13 different scales, therefore even a small false positive rate will result in many false positives).

The image in the detection window is detected as a face when all stages classify the image as a face. A negative outcome of a stage at any point in the cascade immediately rejects the detection window and the next detection window will be processed (see Figure 3.11).

Viola and Jones trained a strong classifier containing only two weak classifiers for the first stage in their cascade. They modified the original

- Given example images $(x_1, y_1), \ldots, (x_n, y_n)$ where $y_i = 0, 1$ for negative and positive examples respectively.
- Initialize weights $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ for $y_i = 0, 1$ respectively, where $m$ and $l$ are the number of negatives and positives respectively.
- For $t = 1, \ldots, T$:
  1. Normalize the weights,
     $$w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$$
     so that $w_t$ is a probability distribution.
  2. For each filter, $j$, train a classifier $h_j$ which is restricted to using a single filter. The error is evaluated with respect to $w_t$:
     $$\epsilon_j = \sum_i w_i |h_j(x_i) - y_i| \tag{3.6}$$
  3. Choose the classifier, $h_t$, with the lowest error $\epsilon_t$.
  4. Update the weights:
     $$w_{t+1,i} = w_{t,i} \beta_t^{1-|h_i(x_i)-y_i|}$$
     where $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$.
- The final strong classifier is:
  $$h(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \theta \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$
  where $\alpha_t = -\log \beta_t$ and $\theta$ is a value in the range [0..1]. $\theta$ is usually set to its original value: $\frac{1}{2}$ (i.e. a democratic vote of all weak classifiers).

**Table 3.2:** The adapted Discrete AdaBoost algorithm. Viola and Jones adapted the original Discrete AdaBoost algorithm by constraining the weak classifier to use only a single filter. $T$ weak classifiers are constructed using the steps defined above. When $T$ weak classifiers are constructed they participate in a weighted vote for the final strong classifier.

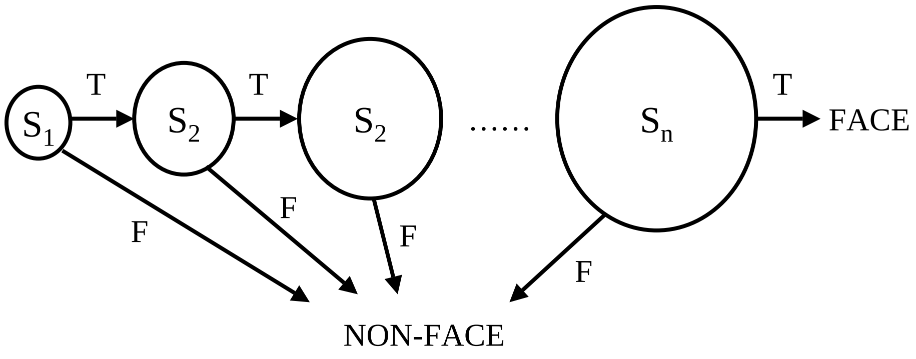


**Figure 3.11:** Schematic overview of the attentional cascade. In order to positively classify the image in a detection window, all the $n$ stages in the cascade will have to classify the image as a face. If a stage rejects the image, the detection window is rejected and the next window will be processed. The first stages are simple and small, eliminating a large number of false positives at little computational effort. The later stages are larger and more complex in order to reduce the false positive rate even further.

threshold of the strong classifier, $\frac{1}{2}\sum_{t=1}^{T}\alpha_t$, in order to achieve a detection rate of 100%. At this threshold setting the false positive rate was 40%, yielding a rejection of 60% of all detection windows that do not contain a face. Since the amount of detection window not containing a face is nearly 100% of all detection windows, practically 60% of all possible detection windows is rejected with an evaluation of only two classifiers.

The stages were trained using a face database containing 4916 faces. The non-face images were gathered from 9500 images that did not contain any faces. For each stage 10 000 non-face images were randomly selected from this set of 9500 images. Viola and Jones used the bootstrapping procedure, as explained in Section 3.2.1, to train the subsequent stages. When a stage was trained, a detector with the partial cascade was run across a large non-face image and a maximum of 6000 false positives were collected. These false positives were then used to train the subsequent stage.

The final detector cascade trained by Viola and Jones contains 32 stages, with a total of 4297 filters. Viola and Jones kept adding more stages to the cascade until the false detection rate on the validation set was nearly zero, while still remaining a high detection rate.

### 3.5.7 Speeding Up The Detector

To increase the speed of the detector at the expense of the accuracy there are three tunable parameters:

1. The **starting scale** is the scale of the detection window when the detection procedure is started. E.g. when the original detection window is $24 \times 24$ pixels and the starting scale is 2.0, then the detector will start with detection windows of size $48 \times 48$. Usually the start scale is set to 1.0. But if the minimum face size should be larger than the original detection window size it is best to use a larger starting scale, since it saves significant computation time.

2. When the detector finishes processing all detection windows of one scale, the detector window is scaled up by the **scale step size**. E.g. when the original detection window is $24 \times 24$ and the scale step is 1.25 then the next detection window size will be 25% bigger: $30 \times 30$ pixels. Viola and Jones obtained good results when the scale step size was set to 1.25.

3. The detector is also scanned across location. After processing a detection window the detection window slides a number of pixels to the next position. The number of pixels it slides depends on the current scale $s$ and the **position step size** $\Delta$. If the current scale is $s$, then the window slides $[s\Delta]$ pixels to the next position, where $[]$ is the rounding operation. When $\Delta > 1$ the detection rate tends to decrease slightly while also the number of false positives decreases.

### 3.5.8 Grouping

The final trained cascade (i.e. detector) is scanned across the whole image at multiple scales and multiple positions. Because the detector is insensitive to small variations in scale and position, usually a large number of detections occur around a face (see Figure 3.12). To incorporate these multiple detections into a single detection, which is more practical, Viola and Jones applied a simple grouping algorithm. This algorithm combines overlapping detection rectangles into a single detection rectangle. Two detections are placed in the same set if their bounding rectangles overlap. For each set the average size and position is determined, resulting in a single detection rectangle per set of overlapping detections.

### 3.5.9 Results

Viola and Jones tested their detector on the MIT+CMU frontal face test set [41] and reported results that are comparative with the systems of Rowley *et. al.* and Sung and Poggio. However the Viola and Jones detector runs

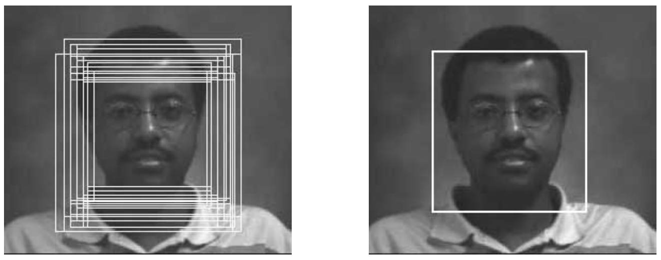

(a) No grouping. (b) Grouping.

**Figure 3.12:** Grouping the multiple detections into a single detection.

much faster (approx. 15 times faster than the Rowley-Baluja-Kanade detector). On average only 8 weak classifiers out of 4297 were evaluated per detection window. The face detector runs at 15 frames per second on a 700 MHz Pentium III processor, when processing images of $384 \times 288$ pixels (starting scale 1.0, scale step size 1.25 and a position step size of 1.5).

### 3.5.10 Related Work

#### Rotated Faces

In [18] Viola and Jones extend their method to be able to detect rotated faces and profile views of faces. Experiments showed that the original five filters used by Viola and Jones were not sufficient to detect non-upright and non-frontal faces with a high accuracy. Therefore Viola and Jones added two extra filters which were able to focus on diagonal features inside the detection window. To detect the rotated and profile faces, Viola and Jones created two face detectors: one for detecting the rotated faces and one for the profile faces. Both detectors use a *pose estimator* in the form of an decision tree. The decision tree runs across the image and returns for each location the pose of the possible face. If the image in the detection window is not a facial image, the returned pose can be considered random. When all poses are estimated, a face detector specialized for a particular pose is run across every estimated pose. This two-stage approach showed good results for both rotated and profile faces. Both face detectors process a $320 \times 240$ image in approximately 0.12-0.14 seconds on a 2.8 GHz Pentium 4. The detection rate of the rotated face detector lies between 89.7% and 95.0%, depending on the number of false positives. The detection rate of the profile face detector lies between 70.4% and 83.1%.

In [24] Lienhart and Maydt enrich the set of filters used by Viola and Jones with 45° rotated filters, in order to detect rotated faces. To compute these rotated filters Lienhart and Maydt created a rotated variant of the integral image. Using this rotated integral image, pixelsums of rotated rectangular areas can be computed in four lookups. Combined with an post-optimization technique, these extended rotated filters performed much better than the original filters used by Viola and Jones. The system of Lienhart and Maydt processes a $320 \times 240$ image in approximately 0.20 seconds on a 2.0 GHz Pentium 4. The detection rate of their rotated face detector lies between the 90.0% and the 95.4%, depending on the false positive rate.

### Gender And Ethnicity Classification

Shakhnarovich *et. al.* present in [36] a framework, based on the work of Viola and Jones, to classify the gender and ethnicity (Asian or non-Asian) of a person. They trained a strong classifier with a few hundred filters. The output of this classifier is a scalar whose sign determines class membership (i.e. male or female and Asian or non-Asian), and the magnitude can be considered as a confidence in the decision. They incorporated this classifier into a novel tracking algorithm. Based on the output of the classifier on the previous images, the tracking algorithm matches persons in the current image. For gender classification they reported error rates between 24.5% and 21.0%, depending on the number of weak classifiers. For the ethnicity classification error rates between 24.3% and 20.8% were reported. With the tracking algorithm these error rates dropped to 10% for gender classification and 17% for ethnicity classification.

### Other Boosting Algorithms

Viola and Jones use an adapted version of the Discrete AdaBoost algorithm. In [23] Lienhart *et. al.* test the performance of the face detector when trained with Discrete AdaBoost, Real AdaBoost and Gentle AdaBoost (the latter two are two variants of the original Discrete AdaBoost algorithm). They conclude that Gentle AdaBoost performs best, despite the fact that on average it required fewer weak classifiers.

AdaBoost is a sequential forward search procedure. When AdaBoost selects a weak classifier during a boosting cycle, this weak classifier is placed in the set of weak classifiers and this set will not be changed. If all possible weak classifiers are evaluated during each boosting cycle this is not a problem, because one can be sure that only the best weak classifiers are selected during each boosting cycle. However if only a limited amount of the possible number of weak classifiers is evaluated, it could be the case that a worse weak classifier is selected. This weak classifier will stay in the set of weak classifiers, possibly dropping the performance of the final strong classifier. This is called *boosting redundancy.* To reduce the boosting redundancy Li *et. al.* invented a floating search variant of the AdaBoost algorithm called FloatBoost [22]. Instead of fixing the set of weak classifiers, FloatBoost uses a backtracking algorithm that allows deletion of those weak classifiers that are non-effective or unfavorable in terms of the error rate. This leads to a strong classifier containing fewer weak classifiers, or achieves lower a error rate with the same number of weak classifiers.

In [48] Xiao *et. al.* propose a very promising boosting method for object detection called the *boosting chain*. This algorithm integrates the boosting method and bootstrap training into a single learning procedure. After each boosting cycle the bootstrap method collects negatives for the next boosting round. To reduce the boosting redundancy a linear optimization scheme is applied, resulting in fewer weak classifiers and lower error rates. Their results on the face detection problem outperform AdaBoost and FloatBoost (i.e. lower false positive rates and higher detection rates on both training and test data).

## 3.6 Discussion Of Face Detection Techniques

In this chapter we discussed five different detection techniques. We decided to implement two of the most promising of these techniques on the INCA$^+$: a skin color based face detector and the Viola and Jones face detector.

The skin color based face detector was chosen because of its simplicity and its popularity among the face detection community. Its simplicity could allow the algorithm to run fully, or at least a large part of it, on Xetal. Part of our objective was to find a face detection technique which could fit on the low-level Xetal chip to relief the TriMedia processor, and the skin color based face detector looks very promising to do such a task.

The Viola and Jones face detector was chosen because of its reported accuracy and the availability of the source code [1]. The detector runs at high speeds while maintaining a high accuracy, making at a robust real-time face detection algorithm. Furthermore the algorithm works on greyscale images and is thus insensitive to color changes and a camera does not need to be equipped with a color sensor, allowing cheaper hardware. Most importantly: the Viola and Jones technique was not designed specifically for face detection like for instance the skin color based face detection. It allows the learning of several different types of objects, making it very suitable as a generic object detector.

In the next chapter we will compare the two implemented face detection techniques. Based on the outcome of that experiment we will decide which of the two techniques suits best on the INCA$^+$.

# Chapter 4

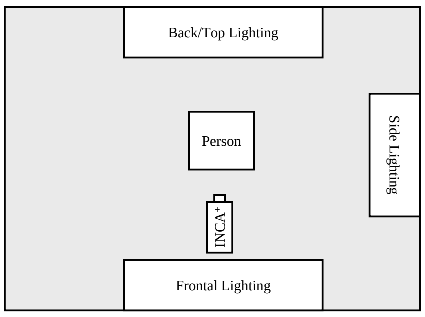
# Face Detection Experiments

In this chapter we will compare a skin color based face detector against the Viola and Jones face detector. Both techniques were successfully implemented on the INCA$^+$. The skin color based face detector uses the ideal dual approach: skin color segmentation (i.e. a large part of the face detection task) is done completely on Xetal. Xetal sends the segmented binary skin image to the TriMedia, which further analyzes the skin image in order to determine the location of the face. The Viola and Jones detector proved to be too complex to run on Xetal and therefore we had to implement it as a whole on the TriMedia processor.

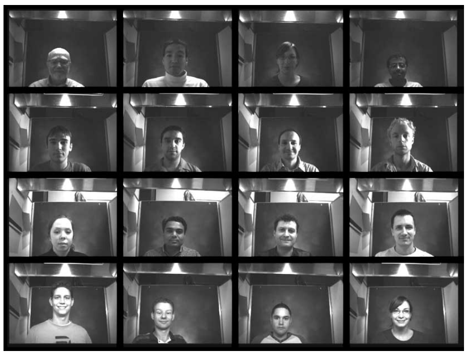
After implementing both face detectors on the INCA$^+$ our next step was to evaluate the performance of both detectors. We did this by comparing the performance of both detectors on a *test database*. Section 4.1 explains how this test database was created. Each image in the test database contains a single face and both face detectors had to determine the position and size of this face. In order to determine whether the outcome of a face detector on a test image was correct or incorrect we created a *detection validation model*, which is described in Section 4.2. In Section 4.3 we tested the skin color based face detector on the test database and demonstrate the results. In Section 4.4 we tested the Viola and Jones face detector on the test database and demonstrate the results. We will conclude this chapter with a discussion on the results of both detectors and we will determine which of the two techniques will be used on our architecture.

## 4.1 Photoshooting Experiment

To compare the two algorithms we set up a photo shooting experiment to create our test database. In total 800 frontal face images were taken of 20 different individuals under four different lighting conditions using the INCA$^+$ itself. We used back/top lighting, side lighting, ambient/environment lighting and frontal lighting as different lighting conditions. The individuals were chosen in such a way that there was enough variety in gender, age and race. Also some persons had a beard and some were wearing glasses. A schematic overview of our photo shooting experiment as well as some of the images we took are shown in Figure 4.1. The images were taken in full color and are $640 \times 480$ pixels in size.



(a) Schematic overview of the photo shooting experiment.



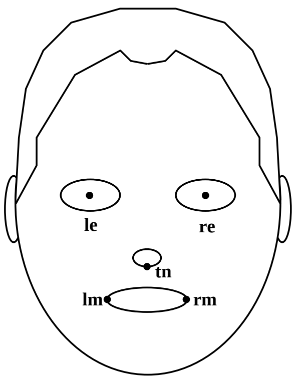
(b) Some of the images we took.

**Figure 4.1:** The physical setup of our photo shooting experiment is presented in (a). We used four different lighting conditions: back/top lighting, side lighting, ambient/environment lighting and frontal lighting. (b) presents some of the images we took. From the top to the bottom row: back/top lighting, side lighting, ambient/environment lighting and frontal lighting respectively.

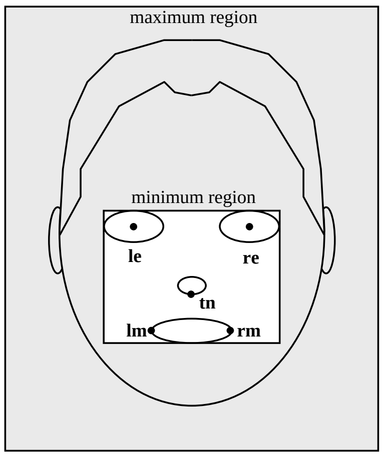
## 4.2 Detection Validation Model

In order to create our detection validation model we first need to define the *ground truth* for each image. We used the ground truth as defined in the MIT+CMU set [41]. In this set faces are manually labeled using five facial feature points: the center of the left and right eye (**le** and **re**), the tip of the nose (**tn**) and the left and right corner of the mouth (**lm** and **rm**) as shown in Figure 4.2(a). Another method for defining the ground truth is to manually define a ground truth rectangle around the face. A big drawback of this method is that it is very sensitive to the perception of what is a face and what is not of the person who is defining the rectangle. Ground truth rectangles usually need to be 'drawn' by hand making it easy to make mistakes, especially when a lot of faces need to be processed by hand. The MIT+CMU approach therefore seems to be a more accurate way to define the ground truth. However these five ground truth points do not define the validity of a detection[1]. For this we created the detection validation model based on the five ground truth points. In our detection validation model two validation rectangles are defined: one specifying the minimum face region and one specifying the maximum face region (see Figure 4.2(b)).

- The minimum region encloses all important facial features. In our experiment these important features are the eyes, the nose and the mouth. A face detection is only valid if all these features are enclosed by the detection rectangle.
- The maximum region ensures an upper bound on the size and position of the detection. A face detection is only valid when the detection rectangle is fully enclosed by the maximum region.



(a) The ground truth.



(b) The detection validation model.

**Figure 4.2:** The ground truth and our detection validation model. In (a) we present the five labeled facial features used as ground truth. These five feature points are also used as ground truth in the MIT+CMU database [41]. In (b) we present our detection validation model. A detection is classified as valid when its detection rectangle fully encloses the minimum region and if and only if it its detection rectangle is fully enclosed by the maximum region. The grey area thus represents a valid area for the detection rectangle.

[1] None of the papers we have studied explain how to discriminate a valid detection from an invalid detection. Therefore a single detection validation model that is used by everyone seems to be lacking in the face detection community. This makes it very hard to compare performance results of different face detectors. The creation and documentation of our validation model could be a first step in the creation of a consistent single validation model.

### 4.2.1 Determining The Validation Regions

To determine the size and location of the two validation regions using only the five ground truth points, we created a facial model first. This model is shown in Figure 4.3 and it is constructed by analyzing some of the faces of our test database and combining these measurements with existing facial models designed by others researchers [25, 6, 35].

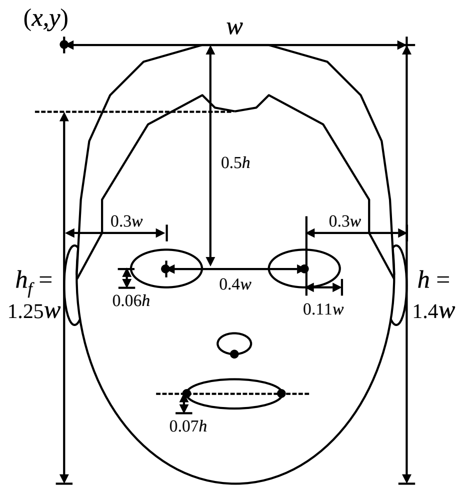


**Figure 4.3:** Our facial model based on a combination of existing facial models and an analysis of some of the faces we captured during our photoshoot experiment.

Using this facial model and the position of the eyes, **le** and **re**, we can compute the bounding rectangle of the face:

$$\begin{aligned} w &= \frac{||\mathbf{le} - \mathbf{re}||}{0.4} \\ h &= 1.4w \\ x &= x_{\text{le}} - 0.3w \\ y &= \frac{y_{\text{le}} + y_{\text{re}}}{2} - 0.5h \end{aligned}$$

where $(x, y)$ specifies the top left corner of the facial bounding rectangle and $w$ and $h$ specify the width and height respectively. Our experiment only uses frontal faces, hence the $y$-position of both eyes should be approximately the same. But to increase the accuracy of the position of the bounding rectangle, the average of the $y$-position of both eyes is used instead of selecting a single $y$-position.

The maximum region was empirically determined at 1.15× the width and height of the facial bounding box. This value ensures that the complete head is included in the region and it allows small variations in size and/or position. Using these metrics we can compute the maximum region by using the following equations:

$$\begin{aligned} x_{\max} &= x - \frac{1.15w - w}{2} \\ y_{\max} &= y - \frac{1.15h - h}{2} \\ w_{\max} &= 1.15w \\ h_{\max} &= 1.15h \end{aligned}$$

As mentioned earlier the minimum region should include at least the eyes, nose and mouth. Using our facial model and the location of the eyes and mouth, we can compute the minimum region:

$$\begin{aligned} x_{\min} &= x_{\mathrm{le}} - 0.11w \\ y_{\min} &= \frac{y_{\mathrm{le}} + y_{\mathrm{re}}}{2} - 0.06h \\ w_{\min} &= (x_{\mathrm{re}} + 0.11w) - x_{\min} \\ h_{\min} &= (\frac{y_{\mathrm{lm}} + y_{\mathrm{rm}}}{2} + 0.07h) - y_{\mathrm{in}} \end{aligned}$$

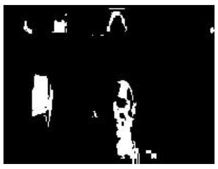
## 4.3 Skin Color Based Face Detection Experiment

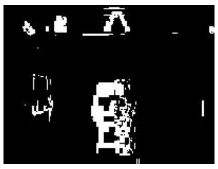
The performance of a skin color based face detector heavily depends on the performance of the chosen skin color segmentation method. If the skin color segmentation method performs well, a good face detector based on that segmentation method could be developed. However if the segmentation fails, it will be very hard to create a good face detector. This skin color experiment therefore tries to determine how various segmentation methods perform.

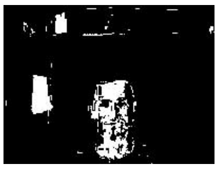
In our experiment we tested four different skin color segmentation methods: an RGB based segmentation method as described in [19], an YCbCr based and an HSI based method both taken from [15] and a method which combines the output of the RGB, YCbCr and HSI segmentation methods. This combined method was inspired by work of Singh *et. al.* in [38]. Our main motivations to choose these techniques was the fact that they are easily implemented on Xetal and they showed good results in the literature [19, 15, 38, 45].

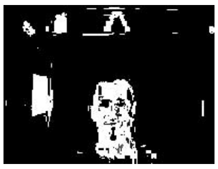
After the skin color segmentation we applied erosion and dilation to remove background noise. For face extraction a contour matching algorithm taken from the OpenCV library [1] was used. This contour matching algorithm finds connected skin pixels and groups them into skin clusters. Since every image only contains a single face we can assume that the largest skin cluster in the image is the face[2]. However in many of the images this largest skin cluster also contains a part of the neck. To remove the neck we disregard the height of the skin cluster, instead we compute the height using our facial model: the height of the face (from chin to hairline) is $1.25\times$ the width of the face. An example of our skin based face detector using the combination segmentation method is shown in Figure 4.4. The output face detection rectangle will be evaluated using the detector validation model. The processing time for a single $640 \times 480$ image ranged from 0.05 to 0.07 seconds (i.e. 15-20 frames per second) on the INCA$^+$.

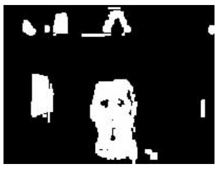
(a) Original

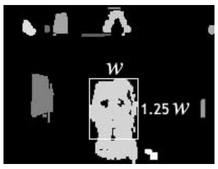

(b) RGB segmentation


(c) YCbCr segmentation


(d) HSI segmentation


(e) Combined segmentation


(f) Erosion/dilation



(g) Find largest contour and determine detection rectangle

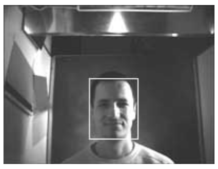

(h) Output detection rectangle superimposed on the original image

**Figure 4.4:** Our skin color based face detector with the combined segmentation methods.

[2] When creating a robust skin color based face detector this approach is not sufficient and a more complex face searching and verification algorithm needs to be incorporated (e.g. with additional eyes and mouth searching and matching).

### 4.3.1 Results

The overall detection rates of our skin based face detector applied on all 800 images using the four different segmentation methods, are shown in Table 4.1[3]. It is clear that when using the combined segmentation method the total detection rate is the highest.

| Segmentation Method | Detection Rate |
|---|---|
| RGB | 42.3% |
| YCbCr | 54.0% |
| HSI | 52.1% |
| Combined | 73.5% |

**Table 4.1:** Overall detection rate on the test database using the skin based face detector with different segmentation methods.

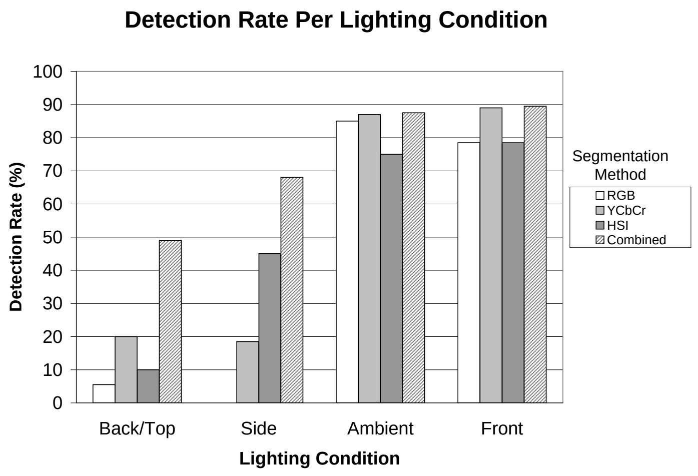


**Figure 4.5:** Detection rate of the skin based face detector using different lighting conditions.

Figure 4.5 shows the detection rate per different lighting condition. The results are good when using the frontal and ambient lighting condition and are comparative with the reported detection results in [38]. None of the individual segmentation methods performs very well when using back/top

[3] Note that only the detection rate is presented. For all images only a single detection rectangle was created, which either correctly or incorrectly classified the face. Therefore the false positive rate is equal to the number of false negative rate (100% − detection rate (in %)).

or side lighting, however when combining the segmentation methods the performance increases significantly in these lighting conditions. But still the performance in the ambient and frontal lighting conditions is much better. The cause of the lower detection rates when using the back/top or side lighting is a worse skin segmentation in these lighting conditions, as shown in Figure 4.6.

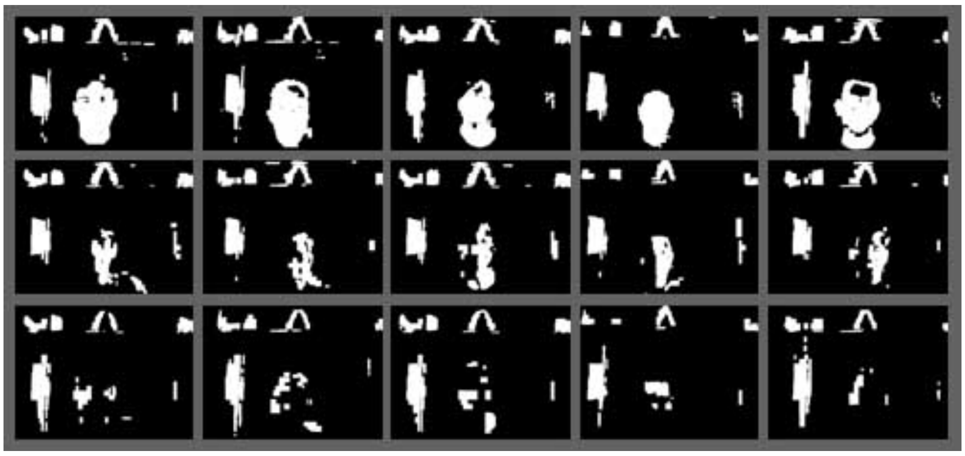


**Figure 4.6:** Skin segmentation examples under different lighting conditions. From the top to the bottom row the same individuals are presented under the following lighting conditions respectively: frontal lighting, side lighting and back/top lighting. These examples clearly demonstrate the cause of the lower detection rates when using back/top or side lighting: the segmentation in these lighting conditions is worse than when using frontal lighting.

## 4.4 Viola And Jones Face Detection Experiment

The performance in terms of speed and accuracy of the Viola and Jones detector depends on the chosen cascade. We tested two setups: a very accurate but slow setup and a less accurate setup which is able to run in near real-time on the INCA$^+$.

For the accurate setup we used the cascade which is part of the OpenCV library. This cascade contains 24 stages and has a total of 2913 weak classifiers. Its starting window size is $24 \times 24$ pixels. The starting scale was set to 1.0, the scale step size was set to 1.1 and the position step size $\Delta$ was set to 1.0. In total 32 different scales were checked, yielding a total of more than 1.8 million possible detection windows. The processing time for a single $640 \times 480$ image ranged from 20 to 25 seconds on the INCA$^+$. This is far from being real-time, but it is a good indication of the upper bound of the accuracy of this algorithm.

The second setup used a smaller cascade trained by Philips Aachen [31]. This cascade contains 24 stages with a total of 651 weak classifiers. Various optimizations were applied to increase the speed of the detector at little expense of the accuracy. The image was downscaled (using the image co-processor) to a size of $320 \times 240$ to reduce the number of possible detection windows. The starting window size is $20 \times 20$ pixels. The starting scale was set to 2.0, the scale step size was set to 1.2 and the position step size $\Delta$ was set to 1.5. In total 10 different scales were evaluated yielding a total of more than 200 000 possible detection windows. The processing time for a single $640 \times 480$ image ranged from 0.20 to 0.25 seconds (i.e. 4-5 frames per second) on the INCA$^+$.

### 4.4.1 Results

Since every image only contains a single face we can collect all detections and average the size and position into a single detection rectangle. We tested both setups on our face database and the result are displayed in Table 4.2.

| Cascade | Detection Rate |
|---|---|
| OpenCV | 93.1% |
| Philips Aachen | 86.8% |

**Table 4.2:** Overall detection rate on the test database using the Viola and Jones face detector with different cascades.

Figure 4.7 shows the detection rate of both detectors under different lighting conditions. The accurate setup performed very good under all lighting conditions. The fast, less accurate setup also performs reasonably good, it only performs slightly worse when using side lighting.

## 4.5 Discussion

In this chapter we compared a skin based face detector against the Viola and Jones face detector. When using ambient or frontal lighting the skin based face detector performs well. However when using side or back lighting the skin segmentation often fails, making it very hard for the face detector to determine the location of the face. Using complex face localization techniques and possibly a better and more complex segmentation method this problem could probably be solved (partially). But the main motivation to choose the skin color technique was its simplicity. When adding complex

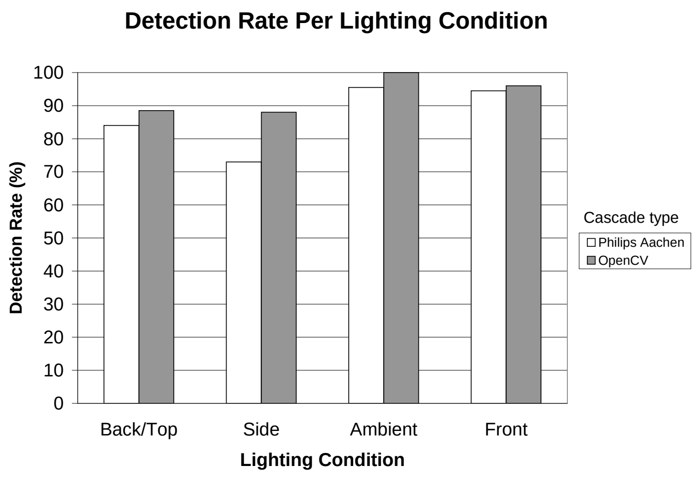


**Figure 4.7:** Detection rate of the Viola and Jones face detector using different lighting conditions.

routines to the skin based face detector to improve the detection accuracy, this simplicity is lost and the performance in terms of speed will drop.

The Viola and Jones algorithm proved to be robust under different lighting conditions. We were also able to implement a reasonable fast detector (i.e. 4-5 frames per second) on the INCA$^+$ while maintaining a high detection rate on our test set. The Viola and Jones method outperformed the skin based method under all lighting conditions, even when the fast and less accurate classifier was used. Furthermore the Viola and Jones method is not constrained solely to face detection. The framework is able to learn and detect other objects as well, only a different cascade needs to be trained. In an embedded architecture with limited resources, such as the INCA$^+$, it is useful to have a single generic object detection framework, instead of several implementations to detect various objects.

We decided, based on the great differences in performance and the ability to learn other objects, to use the Viola and Jones face detector on the INCA$^+$. The only drawback of this method is that it is executed completely on the TriMedia. It would be ideal if the Viola and Jones method could be implemented on Xetal, or a similar SIMD processor. The following chapter will discuss whether it is possible to adapt the algorithm to enable it to run on Xetal.

# Chapter 5

# Viola And Jones Detector On SIMD

The previous chapter showed that the Viola and Jones detector outperforms the skin color based face detector. However this algorithm only makes use of the sequential TriMedia processor and does not benefit at all from the parallel processing power of the SIMD Xetal.

One of our objectives was to investigate the possibility to adapt the chosen face detector so that it could run on the low-level Xetal processor. Unfortunately during our research we found out that it was not possible to implement the Viola and Jones detector on Xetal, simply because Xetal lacks memory and processing power to do such a complex task. However we still wanted to investigate whether it is possible to adapt the Viola and Jones detector to a Xetal like SIMD processor. If it turns out to be possible to create an SIMD version of the Viola and Jones detector, the detector could run on possible future SIMD processors (e.g. the successor of Xetal).

Since the original Viola and Jones algorithm was designed for a sequential processor, a number of modifications need to made in order to adapt the Viola and Jones detector to an SIMD processor. These modifications are presented in this chapter. First the sliding window principle is adapted to SIMD. Subsequently we will discuss the usage of the integral image on SIMD. Using the integral image for filter computation on SIMD is not always feasible or possible (depending on the chosen SIMD architecture), therefore Section 5.3 presents two alternative filter computation methods. Section 5.4 discusses lighting correction on SIMD. Section 5.5 will discuss detection at multiple scales and Section 5.6 considers the use of the attentional cascade on SIMD. We will conclude this chapter with a brief discussion on the presented modifications.

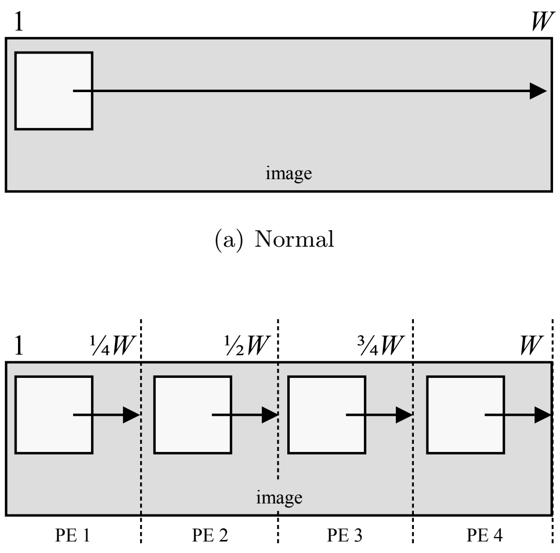
Note that this chapter globally discusses the possibility to implement the Viola and Jones detector on an SIMD processor. More detailed information concerning the number of instructions and memory usage of the algorithm under various SIMD architectures, can be found in Appendix A.

## 5.1 Sliding Windows

The face detector developed by Viola and Jones uses the sliding detection window principle as mentioned in Chapter 3. The algorithm processes a single detection window, slides to a next location and processes the window again. When the attentional cascade is not used, exactly the same processing is done for each detection window. Based on this principle we can exploit SIMD parallelism. Assuming that we have an SIMD processor that operates on a row of pixels[1], each pixel can be considered as a possible detection window location. If there are $P$ processing elements and $W$ pixels in a row (i.e. $W$ = image width), then $P$ detection windows can be processed at the same time. Processing a row is done in $\frac{1}{P}$ of the time it would take for the normal sequential approach. In order to finish processing a whole row of detection windows, each PE will have to process $\frac{W}{P}$ detection windows. For example, if $W = 640$ and $P = 320$, then there are 640 possible detection window locations. Because $P = 320$, 320 detection windows can be processed at the same time. Each PE will only have to process $\frac{W}{P} = 2$ detection windows, instead of 640. An example of this principle is presented in Figure 5.1.

The theoretical maximum speedup is achieved when $W = P$, then the algorithm is $W$-times faster. For example processing a $640 \times 480$ image using the parallel SIMD approach with 640 PEs would take only 1/640 of the time it would take for the normal sequential approach.

[1] i.e. the same topology as the Xetal processor: a row of processing elements processing a row of pixels. If for instance the image width is 640 pixels and there are 320 processors, then each processor processes two columns of pixels. Other topologies are also possible (i.e. a vertical layout, rings, stars or hypercubes), but these topologies are beyond the scope of this thesis.



(a) Normal

(b) SIMD

**Figure 5.1:** In (a) the 'normal' sequential situation is depicted. The sliding detection window processes a window, slides to the next location, processes the window etc. In total $W$ different detection window locations need to be processed. On an SIMD processor all PEs process a detection window at the same time as visualized in (b). In this example there are 4 PEs operating on $W$ pixels. Therefore the algorithm processes a row in $1/4$ of the time it would take for the normal sequential approach.

## 5.2 Integral Image On SIMD

Recall the integral image is defined as the sum of all pixels above and to the left of point $(x, y)$:

$$ii(x,y) = \sum_{j=0}^{x}\sum_{k=0}^{y} I(j,k) \tag{5.1}$$

where $ii(x, y)$ is the value of the integral image at $x, y$ and $I(j, k)$ is the value of the original image at $(i, j)$. Viola and Jones rewrote this equation into the following pair of recurrences:

$$\begin{aligned} r(x,y) &= r(x,y-1) + I(x,y) && (5.2)\\ ii(x,y) &= ii(x-1,y) + r(x,y) && (5.3) \end{aligned}$$

where $r(x, y)$ is the cumulative row sum, $r(x, -1) = 0$ and $ii(-1, y) = 0$. Using these recurrences the value of the integral image at $(x, y)$ can be computed with only two lookups. The cumulative row sum can be computed easily on an SIMD architecture: add the current image value $I(x, y)$ to the cumulative row sum of the row above $r(x, y-1)$. However the calculation of Equation 5.3 is a problem since $ii(x, y)$ requires the integral image value of the left neighbor $ii(x-1, y)$ to be known. On an SIMD processor this left value will only be known after the computation since all PEs execute the same instructions at the same time. To solve this problem Equation 5.1 can be rewritten into the following equations:

$$\begin{aligned} c(x, y) &= \sum_{j=0}^{x} I(j, y) \\ ii(x, y) &= c(x, y) + c(x, y-1) \end{aligned}$$

where $c(x, y)$ is the cumulative column sum[2]. Computing the cumulative column sum is computationally relatively expensive as it requires the summation of a whole row of pixels instead of just a single lookup (see also Section B.1.1). The integral image also consumes a lot of memory and therefore it is not always feasible to use an integral image.

If the integral image is used, filters can be computed in the same fashion as already explained in Section 3.5.2. However the number of instructions that is required to compute the filter depends on the memory architecture of the SIMD processor (e.g. fully or limited connected). More details on this subject can be found in Appendix A.

## 5.3 Filter Computation Using Separable Filters

The previous section showed that computing the integral image on an SIMD architecture is relatively expensive. Therefore this section presents two other methods to compute a filter result. These methods are based on so-called *separable filters*. Using these methods for filter computation saves the number of instructions required to compute the integral image. However filter computation itself is computationally more expensive.

First we will give a brief introduction into separable filters. Subsequently we will demonstrate how separable filters can be computed rapidly on an SIMD architecture. Next we will show that the five filters used by Viola and Jones are separable. We will conclude this section by showing that when using the cumulative row sum, separable filters can be calculated even faster.

[2] i.e. the opposite of the cumulative row sum: it is a row containing the summed $x$-values. Again this term is rather ambiguous, but we used it to be consistent with Viola and Jones' definition of the cumulative row sum.

### 5.3.1 Introduction To Separable Filters

A separable filter is a filter such that it can be written into the following form:

$$F(j,k) = F_x(j) \cdot F_y(k) \tag{5.4}$$

where $F_x$ is a component of $x$-coefficients and $F_y$ is a component of $y$-coefficients. For example:

$$F_x = \begin{bmatrix} -1 & 1 \end{bmatrix} \text{ and } F_y = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

Then filter $F$ would be:

$$F = \begin{bmatrix} -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

Notice that this is not a matrix multiplication. Brackets are used to distinguish this multiplication from matrix multiplications. Unlike matrix multiplication this type of multiplication is commutative ($xy \equiv yx$) [7]:

$$F(j,k) = F_x(j) \cdot F_y(k) \equiv F_y(j) \cdot F_x(k)$$

It is easily shown that not every filter can be separated into two components:

$$F = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

The above filter is impossible to separate into the two components $F_x$ and $F_y$. Setting one element of one of the components to 0 would result in the whole row or column to be set to 0. This is in contradiction with $F$ since every column and row contains a 1.

### 5.3.2 Separable Filters On SIMD

The result of a two-dimensional filter applied on an image is defined as:

$$H(x,y) = \sum_{k=0}^{h-1} \sum_{j=0}^{w-1} F(j,k) \cdot I(x+j, y+k) \tag{5.5}$$

where $H(x,y)$ is the filter result at position $(x,y)$, $w$ and $h$ are the width respectively height of filter $F$, which is applied on image $I$. Computing a two-dimensional filter result requires $w \times h$ instructions on a sequential processor[3]. When $F$ is separable we can substitute Equation 5.4 into Equation 5.5, yielding:

$$H(x,y) = \sum_{k=0}^{h-1} \sum_{j=0}^{w-1} (F_y(k) \cdot F_x(j)) \cdot I(x+j, y+k)$$

[3] A separable filter can be computed in $w+h$ multiplications on an sequential processor, however still $w \times h$ instructions are required to fetch all image values.

$$= \sum_{k=0}^{h-1} F_y(k) \sum_{j=0}^{w-1} F_x(j) \cdot I(x+j, y+k) \tag{5.6}$$

Since this type of multiplication is commutative also the following equation is true:

$$\begin{aligned} H(x,y) &= \sum_{j=0}^{w-1} \sum_{k=0}^{h-1} (F_x(j) \cdot F_y(k)) \cdot I(x+j, y+k) \\ &= \sum_{j=0}^{w-1} F_x(j) \sum_{k=0}^{h-1} F_y(k) \cdot I(x+j, y+k) \end{aligned} \tag{5.7}$$

We can rewrite Equation 5.6 and Equation 5.7 into:

$$H(x,y) = \sum_{k=0}^{h-1} F_y(k) \cdot H_x(x, y+k) \tag{5.8}$$

$$H(x,y) = \sum_{j=0}^{w-1} F_x(j) \cdot H_y(x+j, y) \tag{5.9}$$

where:

$$H_x(x,m) = \sum_{j=0}^{w-1} F_x(j) \cdot I(x+j, m) \tag{5.10}$$

$$H_y(n,y) = \sum_{k=0}^{h-1} F_y(k) \cdot I(n, y+k) \tag{5.11}$$

Equation 5.9 is especially useful in an SIMD environment since every PE in the LPA can compute $H_y$ at the same time, as visualized in Figure 5.2. The computation of $H_y$ by all PEs consumes $h$ instructions[4] and will result in a one-dimensional array of results. To obtain the final filter result $H$, the $x$-component is applied on these results consuming $w$ instructions. Therefore computing the filter result of a two-dimensional separable filter takes $w + h$ instructions on an SIMD architecture, while it would take $w \times h$ instructions on a sequential architecture.

[4] Assuming that every PE processes one column (i.e. the image width (in pixels) is equal to the number of PEs). If the number of PEs is smaller, the number of instructions will be larger.

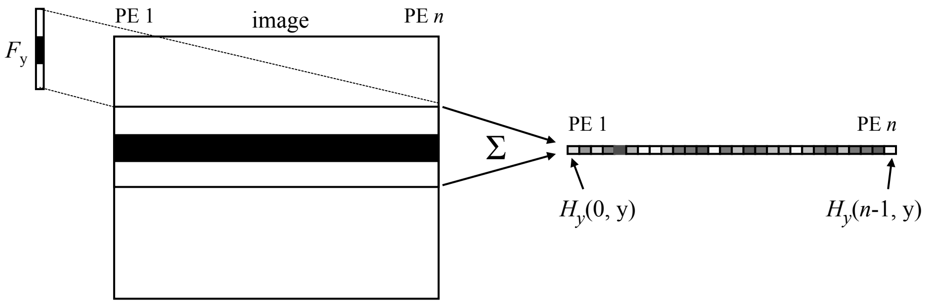


**Figure 5.2:** Applying a vertical filter on an image using SIMD. The vertical filter $F_y$ is applied on the image by all PEs at the same time. Subsequently the results are summed together to form a one-dimensional array of results.

### 5.3.3 Separable Viola And Jones Filters

This section will demonstrate that the filters used by Viola and Jones are separable into an $x$- and $y$ component. We can create the filters by using the following components:

$$\begin{aligned}
F_{x1}(j) &= 1 \\
F_{x2}(j) &= \begin{cases} 1 & \text{if } 0 \le j \le \frac{1}{2}w - 1 \\ -1 & \text{if } \frac{1}{2}w \le j \le w - 1 \end{cases} \\
F_{x3}(j) &= \begin{cases} 1 & \text{if } 0 \le j \le \frac{1}{3}w - 1 \\ -1 & \text{if } \frac{1}{3}w \le j \le \frac{2}{3}w - 1 \\ 1 & \text{if } \frac{2}{3}w \le j \le w - 1 \end{cases} \\
F_{y1}(k) &= 1 \\
F_{y2}(k) &= \begin{cases} -1 & \text{if } 0 \le k \le \frac{1}{2}h - 1 \\ 1 & \text{if } \frac{1}{2}h \le k \le h - 1 \end{cases} \\
F_{y3}(k) &= \begin{cases} 1 & \text{if } 0 \le k \le \frac{1}{3}h - 1 \\ -1 & \text{if } \frac{1}{3}h \le k \le \frac{2}{3}h - 1 \\ 1 & \text{if } \frac{2}{3}h \le k \le h - 1 \end{cases}
\end{aligned}$$

where $F_{x1}$ and $F_{y1}$ are identity components (i.e. always 1). Using these components, the five Viola and Jones filters are defined as follows:

$$\begin{aligned}
F_{\text{h\_edge}}(j,k) &= F_{x1}(j) \cdot F_{y2}(k) \\
F_{\text{v\_edge}}(j,k) &= F_{x2}(j) \cdot F_{y1}(k) \\
F_{\text{h\_line}}(j,k) &= F_{x1}(j) \cdot F_{y3}(k) \\
F_{\text{v\_line}}(j,k) &= F_{x3}(j) \cdot F_{y1}(k) \\
F_{\text{diag}}(j,k) &= F_{x2}(j) \cdot F_{y2}(k)
\end{aligned}$$

This is visualized in Figure 5.3.

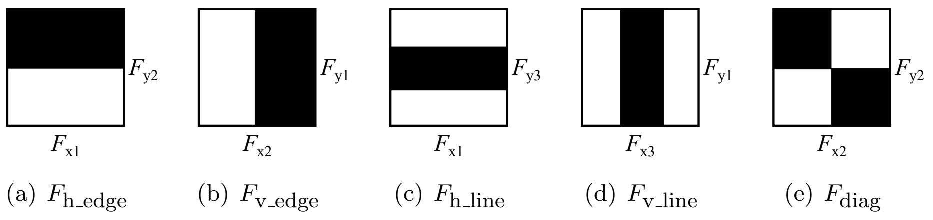


**Figure 5.3:** Viola and Jones filters separated into an $x$- and $y$-component.

### 5.3.4 Cumulative Row Sum

When looking at the five separable Viola and Jones filters, defined in the previous section, it can be observed that the $y$-component can either be $F_{y1}$, $F_{y2}$ or $F_{y3}$. As explained in the Section 5.3.2 computing $H_y$ normally takes $h$ instructions. This section will demonstrate that the computation of $H_{y1}$, $H_{y2}$ and $H_{y3}$ can be performed in much less instructions when using the cumulative row sum. Recall the cumulative row sum is defined as:

$$r(x, y) = \sum_{k=0}^{y} I(x, k) \tag{5.12}$$

and its recursive version is easily implemented on SIMD as already mentioned in Section 5.2. $H_{y1}$ is defined as:

$$H_{y1}(n, y) = \sum_{k=0}^{h-1} F_{y1}(k) \cdot I(n, y + k) \tag{5.13}$$

Since $F_y(k) = 1$ for all $k$, Equation 5.13 can be rewritten into:

$$H_{y1}(n, y) = \sum_{k=0}^{h-1} I(n, y + k) \tag{5.14}$$

Under the assumption that $I(x, -1) = 0$, we can rewrite Equation 5.14 into the following form:

$$H_{y1}(n, y) = \sum_{k=0}^{y+h-1} I(n, k) - \sum_{k=0}^{y-1} I(n, k) \tag{5.15}$$

when combining this equation with the definition of the cumulative row sum this equals:

$$H_{y1}(n, y) = r(n, y + h - 1) - r(n, y - 1) \tag{5.16}$$

Therefore computing $H_{y1}$ consumes 1 instruction (1 subtraction) instead of $h$. This is visualized in Figure 5.4(a). Using this approach we can also rewrite $H_{y2}$ and $H_{y3}$:

$$\begin{aligned} H_{y2}(n, y) \;=\;& \Big(r(n, y + h - 1) - r(n, y + \frac{1}{2}h - 1)\Big) - \\ & \Big(r(n, y + \frac{1}{2}h - 1) - r(n, y - 1)\Big) \end{aligned}$$

Computing $H_{y2}$ consumes 3 instructions (3 subtractions) as visualized in Figure 5.4(b).

$$\begin{aligned} H_{y3}(n, y) \;=\;& \Big(r(n, y + h - 1) - r(n, y + \frac{2}{3}h - 1)\Big) - \\ & \Big(r(n, y + \frac{2}{3}h - 1) - r(n, \frac{1}{3}y - 1)\Big) + \\ & \Big(r(n, y + \frac{1}{3}h - 1) - r(n, y - 1)\Big) \end{aligned}$$

Computing $H_{y3}$ consumes 5 instructions (4 subtractions and 1 addition) as visualized in Figure 5.4(c).

We have shown that on an SIMD architecture the Viola and Jones filters can be computed in $w + 1$, $w + 3$ or $w + 5$ instructions, when using the cumulative row sum. Using the cumulative row sum can speed up filter computation on SIMD significantly. A drawback of the usage of the cumulative row sum is the memory consumption to store the cumulative row sum values. Furthermore it only works with binary filters where the coefficients of the components are either 1 or -1, separable filters on the other hand can have coefficients of any value[5].

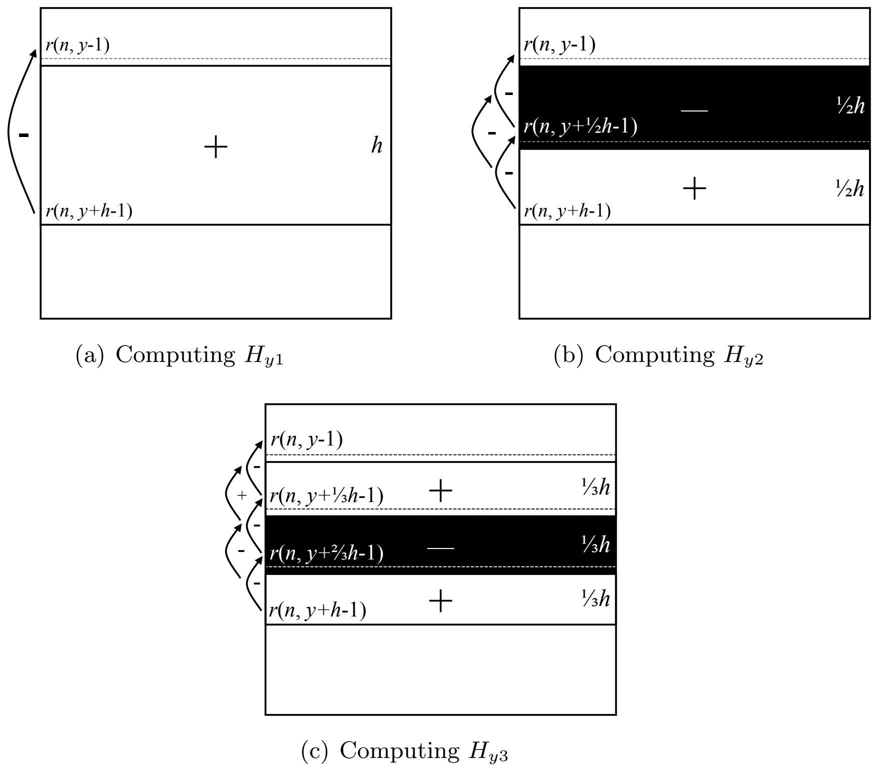
## 5.4 Image Normalization On SIMD

Viola and Jones normalized the image during detection by post multiplying the filter results by the standard deviation $\sigma$ of the image in the detector window. Viola and Jones compute the standard deviation of the pixel values in the detection window by using the following equation:

$$\sigma = \sqrt{\mu^2 - \frac{1}{N}\sum x^2} \tag{5.17}$$

where $\mu$ is the mean, $x$ is the pixel value and $N$ is the total number of pixels inside the detection window. The mean can be computed by computing the pixelsum of the detection window, using the integral image. To

[5] This allows the usage of more complex filters at a slight computational expense. More information on complex filters is presented in Chapter 6.2



(a) Computing $H_{y1}$

(b) Computing $H_{y2}$

(c) Computing $H_{y3}$

**Figure 5.4:** Computing $H_{y1}$ consumes one subtraction. Computing $H_{y2}$ consumes 3 subtractions and $H_{y3}$ 4 subtractions and 1 addition.

calculate $\sum x^2$, Viola and Jones use a squared integral image, which is an integral image only with squared image values. Computing $\sigma$ requires eight instructions for the lookups.

If it is possible to use the integral image on an SIMD architecture, this approach for lighting correction can be adopted. However the number of instructions to compute $\sigma$ might be larger than the eight lookups, depending on the configuration of the architecture (see Appendix A).

We can also compute $\sigma$ by using a separable filter. If we create an *identity filter*: $F_{\text{id}} = F_{x1} \cdot F_{y1}$, and apply it on an image it returns the pixelsum of the area:

$$\begin{aligned} H_{\text{id}}(x,y) &= \sum_{j=0}^{w-1} F_{x1}(j) \sum_{k=0}^{h-1} F_{y1}(k) \cdot I(x+j, y+k) \\ &= \sum_{j=0}^{w-1} \sum_{k=0}^{h-1} I(x+j, y+k) \\ &= \text{pixelsum}(x,y,w,h) \end{aligned}$$

The computation of this separable filter consumes $w + h$ instructions on an SIMD architecture. We can also use the cumulative row sum to compute the identity filter. In that case computing the pixelsum consumes $w + 1$ instructions. To compute the squared pixelsum, $\sum x^2$, we can use the *squared identity filter*:

$$\begin{aligned} H^2_{\text{id}}(x,y) &= \sum_{j=0}^{w-1} F_{x1}(j) \sum_{k=0}^{h-1} F_{y1}(k) \cdot I^2(x+j, y+k) \\ &= \sum_{j=0}^{w-1} \sum_{k=0}^{h-1} I^2(x+j, y+k) \end{aligned}$$

Computing this filter consumes $w + 2h$ instructions. If we use a squared cumulative row sum[6], the squared pixelsum can be computed in $w + 1$ instructions.

Therefore lighting correction on an SIMD processor consumes a minimum of 8 instructions and maximum of $(w + h) + (w + 2h) = 2w + 3h$ per detection window. The actual number of instructions depends on the chosen SIMD architecture.

## 5.5 Detection At Multiple Scales On SIMD

To detect faces at multiple scales Viola and Jones scale up the detection window instead of using the image pyramid approach like for instance Rowley *et. al.* use. The big advantage of the system of Viola and Jones is that processing a single detection window is done in constant time independent of the scale and position of the detection window. Scaling up the detection window rather than scaling down the original image is therefore a logical choice.

However as previous sections have shown, filters are not always computed in constant time on an SIMD architecture. For instance the number of instructions to compute a separable filter depends on the width and height of the filter. Scaling up the filter yields an increase in the number of instructions. To ensure a constant processing time per detection window, we would have to consider the usage of the image pyramid for SIMD architectures that cannot process a detection window in constant time. There is no telling which of the two approaches is the best, it depends on the chosen architecture. More details can be found in Appendix A.

[6] $r^2(x,y) = \sum_{k=0}^{y} I(x,k)^2 = r^2(x, y-1) + I(x,y)^2$

## 5.6 Attentional Cascade On SIMD

The attentional cascade as mentioned in Section 3.5.6 was proposed by Viola and Jones in order to reduce the number of evaluations per detection window. The idea is to use simple classifiers in the first stages to reject the majority of the detection windows, before using more complex classifiers in later stages to achieve low false positive rates.

In our SIMD approach such a cascade can not be used. As mentioned in Section 5.1 the SIMD approach does not process a single sliding window at a time but it processes multiple detection windows at once. When one of the detection windows contains a face, the detector has to process through the whole cascade in order to determine that. But since multiple detection windows at processed once, also the detection windows containing non faces have to process through the whole cascade, rendering the cascade useless.

Therefore our SIMD approach does not use the attentional cascade but it uses a single monolithic strong classifier instead. Viola and Jones trained such a monolithic strong classifier containing 200 weak classifiers and they reported results that were better than when using a 10 stage cascade containing 20 weak classifiers per stage [46].

## 5.7 Discussion

In this chapter we demonstrated that it is possible to adapt the Viola and Jones detector to an SIMD architecture. Because multiple detection windows are processed at the same time, the SIMD version of the algorithm could theoretically be much faster than the original sequential version.

Filter computation and lighting correction can be performed using three different methods: using the integral image, using separable filters and using the cumulative row sum. Because of the great differences between various SIMD architectures (e.g. fully or limited connected, number of PEs etc.) there is no telling which of these methods is the best. If for instance the algorithm runs very fast on a fully connected architecture using one method, it could very well run slow on a limited connected architecture using the same method.

We showed that it is useless to use the attentional cascade on an SIMD architecture. Therefore a single monolithic strong classifier is required. In order to speed up the detection, the number of weak classifiers contained in this strong classifier should be as low as possible, while maintaining a high performance. The next chapter will present two methods which could reduce the number of weak classifiers significantly.

# Chapter 6

# Weak Classifier Improvements

As demonstrated in the previous chapter, it is possible to adapt the Viola and Jones algorithm to an SIMD architecture. However as the attentional cascade cannot be used, every weak classifier contained in the strong classifier needs to be checked for each detection window. Therefore we need to search for methods that reduce the number of weak classifiers significantly.

This chapter will present two methods that improves the performance of weak classifiers. When the individual weak classifiers perform better, the total number of weak classifiers in the strong classifier could be smaller to achieve the same performance.

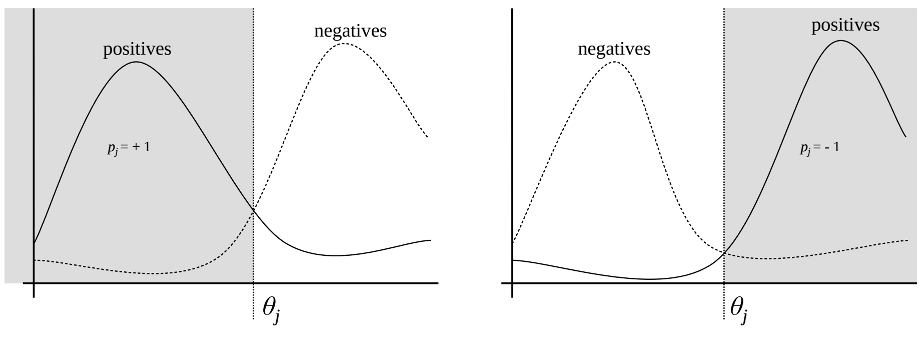
The first improvement, presented in Section 6.1, demonstrates that using multiple thresholds for the weak classifier can improve the performance of the weak classifier significantly. In Section 6.2 the weak classifiers contain more complex filters than the simple Haar-like filter used by Viola and Jones. It demonstrates the effectiveness of these filters and how they could be implemented on an SIMD architecture.

## 6.1 Multiple Threshold Weak Classifiers

In the original AdaBoost algorithm a number of single filter weak classifiers are trained per strong classifier. Recall the weak classifier as it is used by Viola and Jones (see Section 3.5):

$$h_j(x) = \begin{cases} 1 & \text{if } p_j f_j(x) < p_j \theta_j \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

where $f_j$ is the filter, $\theta_j$ is the threshold, $p_j$ is the parity which indicates the direction of the equality sign as shown in Figure 6.1, and $x$ is the input image. For each weak classifier $h_j$ an optimal threshold $\theta_j$ and a parity $p_j$ are determined such that a minimum of positive and negative examples



(a) $p_j = +1$ (b) $p_j = -1$

**Figure 6.1:** The effect of parity $p_j$. In (a) the majority of the positive examples lie left of threshold $\theta_j$, thus $h_j(x) = 1$ if $f_j(x) < \theta_j$ and $p_j = +1$. In (b) the majority of the positive examples lie right of threshold $\theta_j$, thus $h_j(x) = 1$ if $f_j(x) > \theta_j \equiv -f_j(x) < -\theta_j$ and $p_j = -1$.

are misclassified [46]. Therefore the probability of making an error (misclassification) should be minimal. According to Bayes decision theory the probability of making an error is defined as:

$$p(\text{error}) = p(\text{error}|\omega_p)p(\omega_p) + p(\text{error}|\omega_n)p(\omega_n)$$

where $\omega_p$ is the positive (face) class, $\omega_n$ the negative (non-face) class and

$$p(\text{error}|\omega_p) = \int_{\Omega_n} p(\mathbf{x}|\omega_p)d\mathbf{x} \text{ and } p(\text{error}|\omega_n) = \int_{\Omega_p} p(\mathbf{x}|\omega_n)d\mathbf{x}$$

where $\Omega_p$ defines a positive range and $\Omega_n$ defines a negative range as visualized in Figure 6.2. Both the positive and negative examples are normalized:

$$\int_{-\infty}^{\infty} p(\mathbf{x}|\omega_p)p(\omega_p)d\mathbf{x} = \int_{-\infty}^{\infty} p(\mathbf{x}|\omega_n)p(\omega_n)d\mathbf{x} = 1 \ .$$

Because nothing can be said about the *a priori* probabilities $p(\omega_p)$ and $p(\omega_n)$, there is no predetermined probability that an image is a face or not[1], we assume that $p(\omega_p) = p(\omega_n) = \frac{1}{2}$. The probability of making an error is then defined as:

$$p(\text{error}) = \frac{p(\text{error}|\omega_p) + p(\text{error}|\omega_n)}{2} \tag{6.2}$$

[1] In the “real world” the a priori probability of a facial image is much lower than the a priori probability of a nonface image. It would be an interesting research topic to investigate the performance of a weak classifier when a lower a priori probability of the positive facial images is chosen, however this is beyond the scope of this thesis.

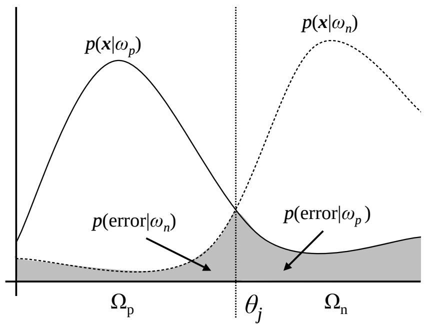


**Figure 6.2:** The errors $p(\text{error}|\omega_p)$ and $p(\text{error}|\omega_n)$.

And this error is minimal at when the threshold $\theta$ is placed at the crossing point of the positive and negative examples (Bayes classifier). Thus when:

$$\begin{aligned} p(\mathbf{x}|\omega_p)p(\omega_p) &= p(\mathbf{x}|\omega_n)p(\omega_n) \overset{p(\omega_p)=p(\omega_n)}{\Longleftrightarrow} \\ p(\mathbf{x}|\omega_p) &= p(\mathbf{x}|\omega_n) \end{aligned} \tag{6.3}$$

If the positive and negative examples can be well separated into two areas as shown in Figure 6.2, a single threshold combined with a parity ensures the lowest possible error. However this makes the assumption that the distribution of positive and negative examples is restricted to identical classes (e.g. the distribution of positive and negative examples can be modeled with a Gaussian with equal $\mu$ and $\sigma$). However if the classes are not identical as shown in Figure 6.3, more than one threshold is required to ensure the lowest error. Using a single threshold therefore does not always result in the lowest possible error, and this could influence the performance of the weak classifier and therefore also the performance of the final strong classifier.

We developed an algorithm that uses multiple thresholds to overcome this dilemma. First the following definitions are made:

$$P(\Omega_{i,j}) = \frac{\int_{\theta_i}^{\theta_j} p(\mathbf{x}|\omega_p)p(\omega_p)d\mathbf{x}}{\int_{-\infty}^{\infty} p(\mathbf{x}|\omega_p)p(\omega_p)d\mathbf{x}} \times 100\% \tag{6.4}$$

$$N(\Omega_{i,j}) = \frac{\int_{\theta_i}^{\theta_j} p(\mathbf{x}|\omega_n)p(\omega_n)d\mathbf{x}}{\int_{-\infty}^{\infty} p(\mathbf{x}|\omega_n)p(\omega_n)d\mathbf{x}} \times 100\% \tag{6.5}$$

where $\Omega_{i,j}$ defines a range enclosed between the two thresholds: $\theta_i$ and $\theta_j$, $i \leq j$. $P(\Omega_{i,j})$ and $N(\Omega_{i,j})$ thus represent the percentage of positive and negative examples in range $\Omega_{i,j}$ respectively. A range $\Omega_{i,j}$ is defined as positive if $P(\Omega_{i,j}) - N(\Omega_{i,j}) > 0$, negative otherwise.

Using these definitions we present our algorithm in Table 6.1. If thresholds are placed at every crossing point between the positive and negative

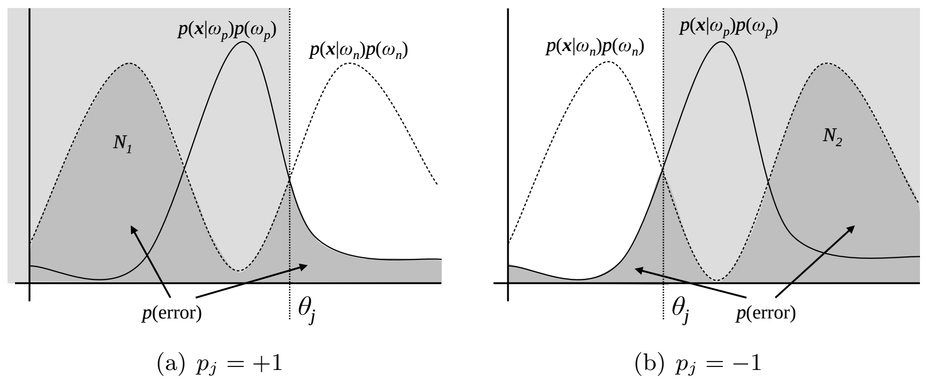


**Figure 6.3:** The single threshold dilemma. No matter how the threshold $\theta_j$ and parity $p_j$ are set, they cannot prevent the negative examples $N_1$ or $N_2$ from being included, yielding a higher $p$(error).

examples, the lowest error is ensured (Bayes classifier). However when the distribution of positive and negative examples fluctuates a lot, this will result in an impractical amount of thresholds. In order to reduce the amount of thresholds we introduce value $T$. For a range $\Omega_{i,j}$ to appear in the set of output ranges $\mathbf{R}$, $P(\Omega_{i,j}) - N(\Omega_{i,j})$ should be larger than $T$. This is visualized in Figure 6.4.

1. Create an empty set of output ranges $\mathbf{R} = \{\}$.
2. Apply a filter $f_j$ on all positive and negative samples to create a filter output distribution of positive and negatives examples.
3. Place thresholds at all crossing points of the positive and negative examples (Bayes classifier). If $C$ thresholds are placed, $C + 1$ ranges will be defined.
4. Find a single positive range $\Omega_{i,j}$ for which:
   - $P(\Omega_{i,j}) - N(\Omega_{i,j}) > T$, where $T \geq 0$.
   - $P(\Omega_{i,j}) - N(\Omega_{i,j})$ is maximal.
   - range $\Omega_{i,j}$ does not overlap with any of the ranges in $\mathbf{R}$.
5. If such a range $\Omega_{i,j}$ exists, add it to the set of output ranges $\mathbf{R}$, and goto step 4. Else goto step 6.
6. Expand all positive ranges by merging each positive range with adjacent ranges and goto step 4. If none of the positive ranges can be further expanded and no more range can be added to $\mathbf{R}$, stop.

**Table 6.1:** Placing the multiple thresholds.

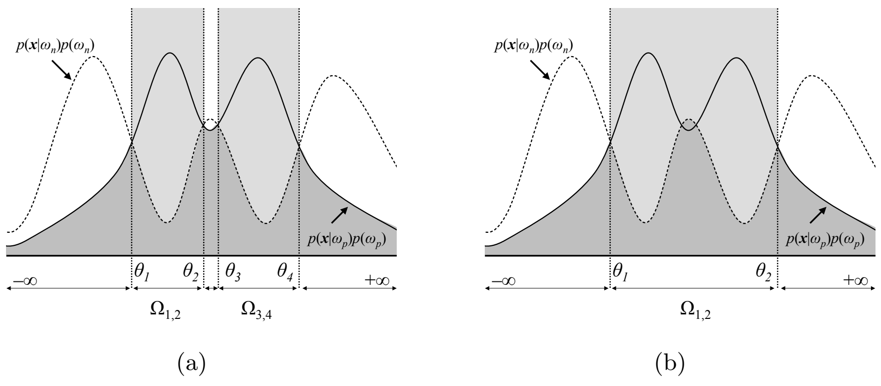


**Figure 6.4:** Expanding the positive ranges to reduce the number of thresholds. In (a) thresholds are placed at every crossing point between the positive and negative examples (Bayes classifier). Therefore the total error (dark grey area) is as low as possible. The light grey areas represent the positive ranges. To reduce the number of thresholds we introduce the value $T$. Suppose we choose a value $T$ and $P(\Omega_{1,2}) - N(\Omega_{1,2}) \leq T$ and $P(\Omega_{3,4}) - N(\Omega_{3,4}) \leq T$. Then our algorithm expands the ranges (step 6) and find the range $\Omega_{1,4}$ and $P(\Omega_{1,4}) - N(\Omega_{1,4}) > T$. Now two thresholds are removed at the expense of a slight increase of the total error.

The original threshold weak classifier (Equation 6.1) can be rewritten into a multiple threshold form as follows:

$$h_j(x) \quad = \quad \begin{cases} 1 & \text{if } \exists \Omega_{k,m} \in \mathbf{R} \text{ such that } f_j(x) \in \Omega_{k,m} \\ 0 & \text{otherwise} \end{cases} \tag{6.6}$$

When looking at Figure 6.5, we can define the error (in %) for the positive and negative class as follows:

$$\begin{aligned} p(\text{error}|\omega_p) &= 100 - \sum_{i=1}^{||\mathbf{R}||} P(\mathbf{R}[i]) \\ p(\text{error}|\omega_n) &= \sum_{i=1}^{||\mathbf{R}||} N(\mathbf{R}[i]) \end{aligned}$$

Using these definitions and Equation 6.2 we can compute the total error, $p(\text{error})$, for the multiple threshold classifier.

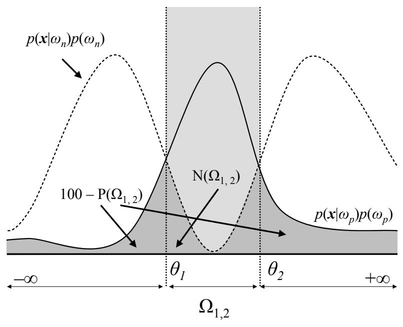


**Figure 6.5:** The total error $p(\text{error})$ of the multiple threshold weak classifier.

### 6.1.1 Experiments And Results

To compare the performance of the multiple threshold weak classifier against the single threshold weak classifier, we set up two experiments. The first experiment was used to determine the optimal $T$ for the multiple threshold weak classifier. The second experiment will then compare results of the multiple threshold weak classifier using this optimal $T$ against the original single threshold weak classifier.

In both experiments we used the MIT+CBCL [2] frontal face database for the training and test data. This database contains a training set, with 2429 faces and 4548 non-faces, and a test set with 472 faces and 23 573 non-faces. All images are cropped and rescaled to a size of $19 \times 19$ pixels. We manually normalized the contrast of the images to minimize the effect of different lighting conditions.

### Determining $T$

In order to determine the optimal $T$, we tested the following eight values for $T$: 0.05%, 0.1%, 0.5%, 1%, 5%, 10%, 20% and 30%. For each different value, 10 strong classifiers were trained. The number of weak classifiers in a strong classifier was set to 50. Each boosting round the weak classifier with the lowest error rate was selected from a pool of 200 random weak classifiers. A fixed random seed was used to ensure that the exact same random weak classifiers were generated and evaluated for all tests.

The average performance of the strong classifiers on the training data and test data is presented in Table 6.2.

| $T(\%)$ | **DR Train** | **FPR Train** | **DR Test** | **FPR Test** | **# Thr.** |
|---|---|---|---|---|---|
| 0 | 100 $\pm$0.03 | 0.22 $\pm$0.08 | 64.3 $\pm$2.5 | 0.98 $\pm$0.18 | 167 $\pm$6.5 |
| 0.05 | 100 $\pm$0.03 | 0.20 $\pm$0.09 | 65.5 $\pm$3.7 | 0.66 $\pm$0.14 | 77.7 $\pm$31 |
| 0.1 | 99.9 $\pm$0.08 | 0.44 $\pm$0.13 | 66.1 $\pm$7.5 | 0.62 $\pm$0.09 | 33.3 $\pm$9.9 |
| 0.5 | 99.7 $\pm$0.13 | 0.54 $\pm$0.16 | 67.0 $\pm$5.8 | 0.54 $\pm$0.13 | 6.1 $\pm$8.7 |
| 1 | 99.7 $\pm$0.13 | 0.43 $\pm$0.10 | 69.1 $\pm$5.1 | 0.41 $\pm$0.10 | 2.9 $\pm$0.12 |
| 5 | 99.7 $\pm$0.13 | 0.51 $\pm$0.09 | 70.4 $\pm$6.0 | 0.53 $\pm$0.10 | 2.7 $\pm$0.10 |
| 10 | 99.7 $\pm$0.16 | 0.57 $\pm$0.16 | 75.5 $\pm$4.3 | 0.58 $\pm$0.11 | 2.6 $\pm$0.09 |
| 20 | 99.0 $\pm$0.28 | 0.89 $\pm$0.20 | 66.9 $\pm$5.6 | 0.66 $\pm$0.17 | 2.2 $\pm$0.15 |
| 30 | 98.2 $\pm$0.40 | 1.02 $\pm$0.19 | 59.3 $\pm$6.6 | 0.79 $\pm$0.09 | 2.0 $\pm$0.04 |

**Table 6.2:** Average performance of the multiple threshold weak classifier experiments using different $T$ settings. DR is the detection rate and FPR is the false positive rate, both measured in percentages. # Thr. is the average number of thresholds.

When looking at Table 6.2 the following observations can be made:

- When $T$ is chosen too small (e.g. $\leq 0.05$) the performance of the final strong classifier on the training data is maximal: 100% detection rate and the lowest false positive rate. However the performance on the test data is better when a larger $T$ is chosen. This could be an indication of a better generalization. When $T$ is chosen too small an impractical amount of thresholds needs to be checked.

- When $T$ is chosen too large (e.g. $\geq 20\%$) the performance of the final strong classifier on the test data will be worse than when a lower $T$ is chosen. $T$ was introduced to reduce the number of thresholds, but a value of $T \geq 20\%$ means that the difference between the percentage of positive examples and negative examples is $\geq 20\%$. Filters that are able to realize such a high difference between the positive and negative examples are usually small filters that tend to focus on small high variance spots (e.g. noise). These small filters are very poor in generalization and therefore the performance drops when $T$ is set too large.

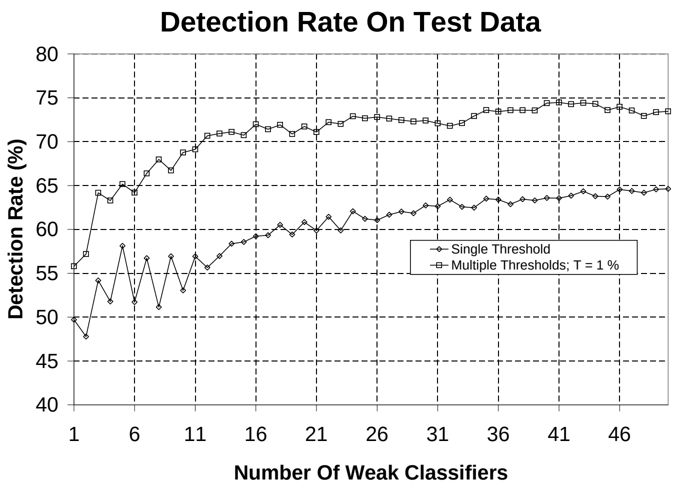
Our optimal value $T$ was chosen by applying the following criterions:

1. The average number of thresholds should not exceed 4, otherwise an impractical amount of thresholds needs to be checked for each weak classifier.
2. The false positive rate on the test data should be the lowest.

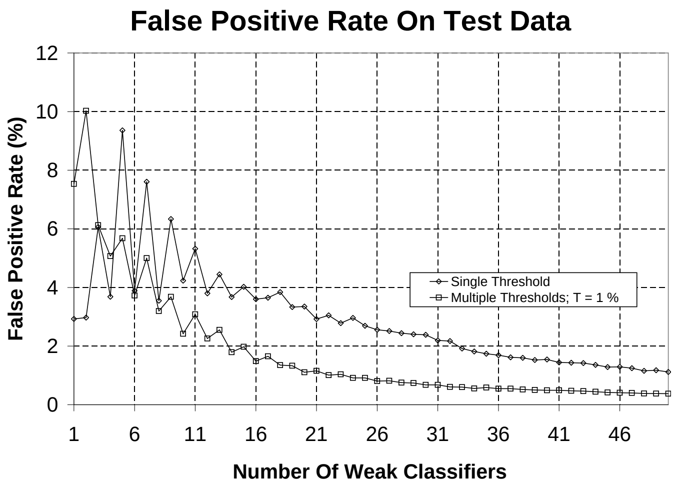
In this case the lowest false positive rate was found when $T = 1\%$. Note that other criterions can also be chosen, for example another criterion could be the highest detection rate on the test data. In that case $T = 10\%$ would be the optimal value.

**Comparing Single And Multiple Threshold Weak Classifiers.**

In the second experiment we trained 20 strong classifiers using the original single threshold weak classifier, and 20 strong classifiers using multiple threshold weak classifiers with the optimal value $T = 1\%$. The number of weak classifiers in the strong classifiers was set to 50. Each boosting round the weak classifier with the lowest error rate was selected from a pool of 500 random weak classifiers. The average performance of the single and multiple threshold approaches are displayed in the figures on the following pages.

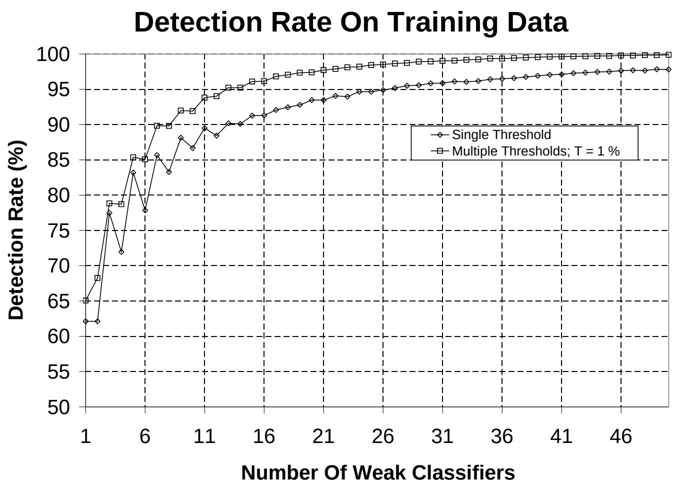


(a)

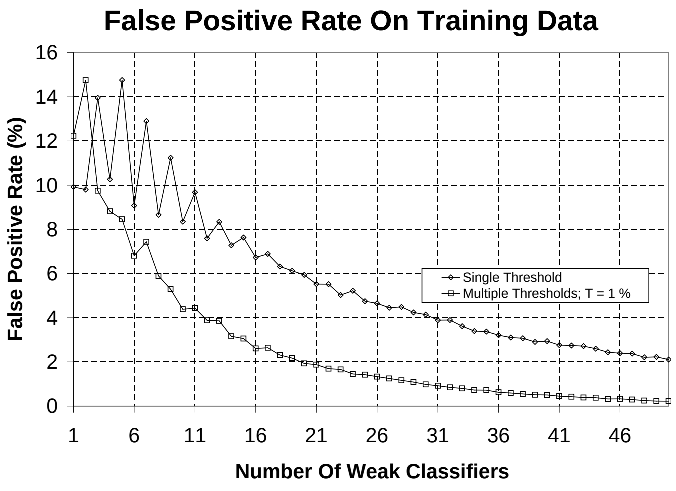


(b)

**Figure 6.6:** Detection rate and false positive rate on test data using single and multiple threshold weak classifiers. When using multiple threshold weak classifiers the detection rate is always higher than when using single threshold weak classifiers. When using 7 multiple threshold weak classifiers, the detection rate is already higher than when using 50 single threshold weak classifiers. More importantly the false positive rate on the test set converges much faster towards zero. When using 20 multiple threshold weak classifiers, both the detection rate and the false positive rate are better than when using 50 single threshold weak classifiers. A reduction in number of classifiers of 250%.



(a)



(b)

**Figure 6.7:** Detection rate and false positive rate on training data using single and multiple threshold weak classifiers. The detection rate is only slightly higher when using a strong classifier containing multiple threshold weak classifiers. The false positive rate however converges much faster towards zero when using multiple thresholds. Notice that when using multiple threshold weak classifiers the oscillations in detection rate and false positive rate are more damped, an indication of a more stable learner.

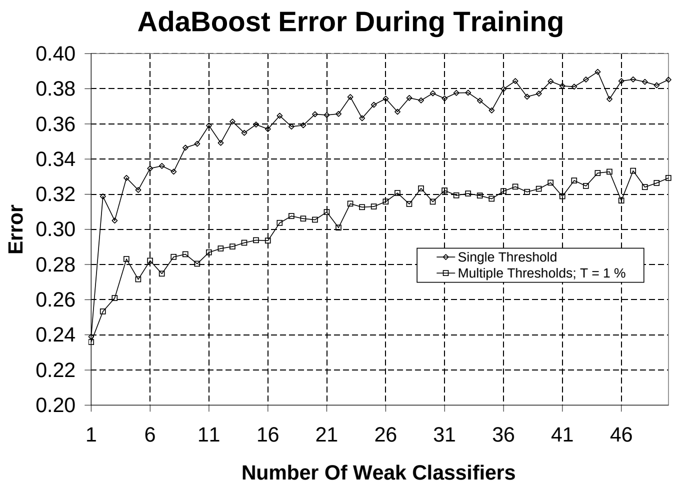


**Figure 6.8:** The AdaBoost error $\epsilon_t$ during training using single and multiple threshold weak classifiers. This figure shows that the AdaBoost error is much lower when using multiple threshold weak classifiers. An indication of a better generalizing weak classifier (i.e. it makes fewer mistakes).

The previous figures demonstrated better results in terms of detection rate and false positive rate, on both training and test set, when using the multiple threshold weak classifiers. To further compare the two approaches, we computed an ROC curve of each strong classifier. We can compute an ROC curve by adjusting the threshold $\theta$ of the strong classifier from 0 to 1 (see Equation 3.7). The averaged ROC curves of the strong classifiers using single and multiple threshold weak classifiers are displayed in Figure 6.9.

### 6.1.2 Discussion

In this section we proposed the idea to use multiple threshold weak classifiers instead of using the single threshold classifiers as used by Viola and Jones. We showed that the performance in terms of detection rate and false positive rate is much better when using multiple threshold weak classifiers. Using this approach might reduce the number of weak classifiers in the final strong classifier significantly at the expensive of a few extra threshold evaluations. The preliminary results look promising, however we should stress that more research is required to obtain the final results. We only used a single facial database with relatively few examples, 2429 positives and 4548 negatives, as compared to the number of examples Viola and Jones use: 4916 positives and 10 000 negatives combined with a bootstrapping algorithm. What the effect of the multiple threshold classifier will be when using other facial databases will have to be determined.

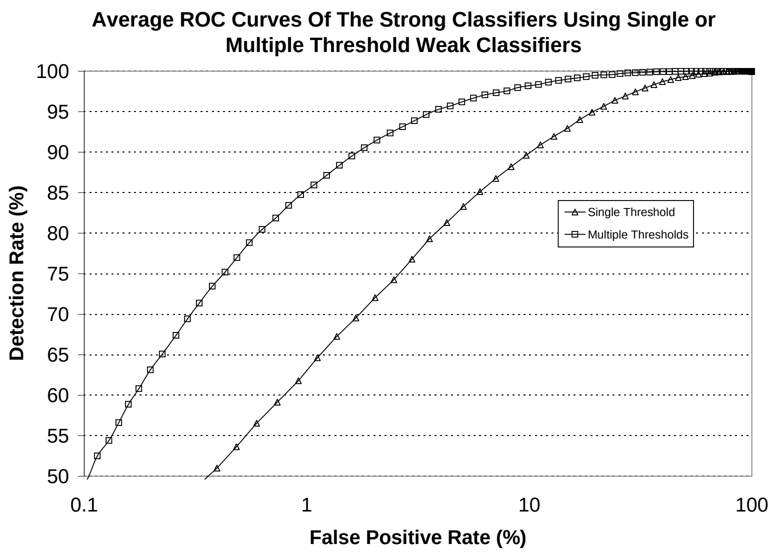


**Figure 6.9:** The averaged ROC curves of the strong classifiers using single and multiple threshold weak classifiers. Notice that the $x$-axis is presented in log domain. This better visualizes the difference between the two curves.

## 6.2 The Effect Of More Complex Filters

In this section we will introduce two new filter sets. These filters are more complex than the original filters used by Viola and Jones. We will determine whether these filters are more effective in the face detection task. If they are, the number of weak classifiers might be reduced significantly.

First we will introduce a slightly more complex filter set based on the original Viola and Jones filters. Next we will introduce a complex filter set based on separable Gabor filters. In Section 6.2.3 we setup an experiment to test the performance of these complexer filter sets when used for the face detection task. We will conclude this section with a brief discussion on the performance of the complex filters.

### 6.2.1 Extended Viola And Jones Filters

Part of our research focused on the question whether more complex Haar-like filters would increase the performance of the weak classifier. Therefore we decided to enrich the original filter set used by Viola and Jones with three new filters as shown in Figure 6.10. The three new filters are:

$$\begin{aligned}F_{x2y3}(j,k) &= F_{x2}(j)\cdot F_{y3}(k)\\ F_{x3y2}(j,k) &= F_{x3}(j)\cdot F_{y2}(k)\\ F_{x3y3}(j,k) &= F_{x3}(j)\cdot F_{y3}(k)\end{aligned}$$

These filters are build of the same components as the original Viola and Jones filters, therefore they can be computed fast using the cumulative row sum (as explained in Section 5.3.4).

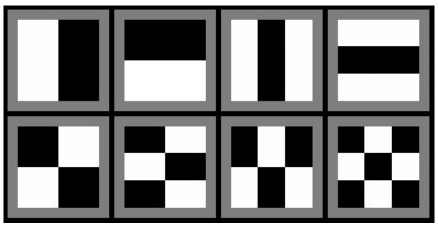


**Figure 6.10:** The extended Viola and Jones set used in our experiment.

### 6.2.2 Gabor Filters

Gabor filters, designed by Dennis Gabor [14], are known for their image or information compaction capabilities. They have been used in image processing applications such as texture segmentation [12], face recognition [47], fingerprint recognition [17] and other areas. The use of Gabor filters is also biologically motivated since the receptive fields of so called simple cells in the primary visual cortex of mammals can be approximated with Gabor functions [32]. The original complex one-dimensional Gabor filter is a modulation of a normalized Gaussian kernel and a complex term [14, 51]:

$$g(t|\sigma,f) = \underbrace{\frac{1}{\sqrt{2\pi\sigma}}e^{\frac{-t^2}{2\sigma^2}}}_{\text{Gaussian envelope}} \cdot \underbrace{e^{i2\pi ft}}_{\text{Modulation term}} \tag{6.7}$$

where $f$ is the frequency of the sinusoidal plane wave.

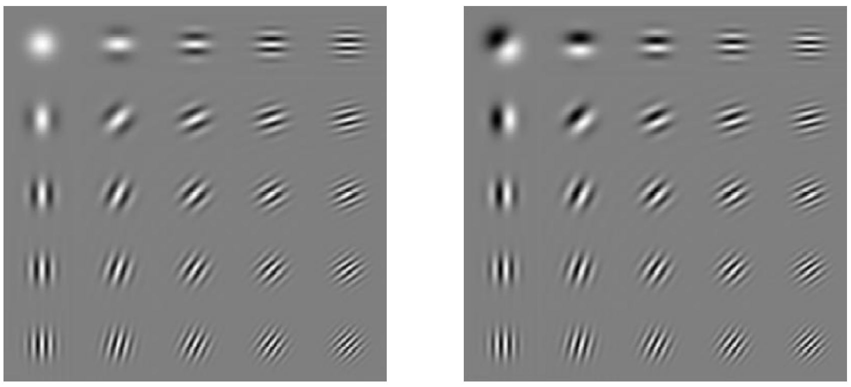
In this section we will try to determine whether the usage of Gabor filters as filters for our face detection method will improve the performance of the detector.

## 2-D Gabor Filters

Daugman extended the 1-D Gabor filter to model the spatial summation properties (of the receptive fields) of simple cells in the visual cortex of mammals, to the following 2-D form [11, 17]:

$$\begin{aligned} G(x,y|\sigma'_x,\sigma'_y,f,\theta) &= e^{-\frac{1}{2}(\frac{x'^2}{\sigma'^2_x}+\frac{y'^2}{\sigma'^2_y})} \cdot e^{i2\pi f x'} && (6.8) \\ x' &= x\sin\theta + y\cos\theta && (6.9) \\ y' &= x\cos\theta - y\sin\theta && (6.10) \end{aligned}$$

where $f$ is the frequency of the sinusoidal plane wave along the direction $\theta$ with respect to the $x$-axis. $\sigma'^2_x$ and $\sigma'^2_y$ are the standard deviations of the Gaussian envelope along the $x'$ and $y'$ axes respectively. A representation of even (real) and odd (complex) 2-D Gabor filters is presented in Figure 6.11.



(a) Even 2-D Gabor Filters (b) Odd 2-D Gabor Filters

**Figure 6.11:** A representation of even an odd 2-D Gabor filters using different orientations and frequency values (taken from [21]).

### Separable 2-D Gabor Filters

To be able to use the 2-D Gabor filters on an SIMD architecture, the filters need to be separable (see Section 5.3.2). The 2-D Gabor filter as defined in Equation 6.8 can only be separated when there is no dependency between $x$ and $y$. Using Equation 6.16 and 6.16 it is seen that this is only the case when $\theta = \{0, \frac{1}{2}\pi, \pi, 1\frac{1}{2}\pi\}$ (i.e. the filter is horizontal or vertical). If $\theta = 0$ or $\pi$ then $x' = y$ and $y' = x$. Equation 6.8 can then be rewritten as follows:

$$G(x, y|\sigma_x, \sigma_y, f, \theta_{\{0,\pi\}}) = e^{-\frac{1}{2}(\frac{x^2}{\sigma_x^2}+\frac{y^2}{\sigma_y^2})} \cdot e^{i2\pi fy} \tag{6.11}$$

$$G(x, y|\sigma_x, \sigma_y, f, \theta_{\{0,\pi\}}) = \underbrace{e^{-\frac{x^2}{2\sigma_x^2}}}_{\text{Gauss}} \cdot \underbrace{\left(e^{-\frac{y^2}{2\sigma_y^2}} \cdot e^{i2\pi fy}\right)}_{\text{1-D Gabor}} \tag{6.12}$$

Equation 6.12 is of the form $F(x, y) = F_x(x) \cdot F_y(y)$, $F_x$ being a Gaussian envelope and $F_y$ being the original 1-D Gabor filter as defined in Equation 6.7 (without the normalization factor). Equation 6.12 is called the horizontal 2-D Gabor filter.

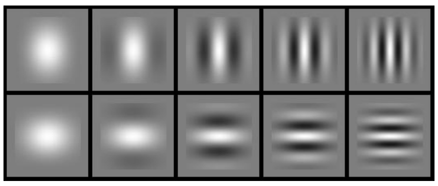
If $\theta = \frac{1}{2}\pi$ or $1\frac{1}{2}\pi$ then $x' = \pm x$ and $y' = \pm y$, then Equation 6.8 can be rewritten as follows:

$$G(x, y|\sigma_x, \sigma_y, f, \theta_{\{\frac{1}{2}\pi,1\frac{1}{2}\pi\}}) = e^{-\frac{1}{2}(\frac{x^2}{\sigma_x^2}+\frac{y^2}{\sigma_y^2})} \cdot e^{i2\pi f\cdot(\pm x)} \tag{6.13}$$

$$G(x, y|\sigma_x, \sigma_y, f, \theta_{\{\frac{1}{2}\pi,1\frac{1}{2}\pi\}}) = \underbrace{\left(e^{-\frac{x^2}{2\sigma_x^2}} \cdot e^{i2\pi f\cdot(\pm x)}\right)}_{\text{1-D Gabor}} \cdot \underbrace{e^{-\frac{y^2}{2\sigma_y^2}}}_{\text{Gauss}} \tag{6.14}$$

Equation 6.14 is of the form $F(x, y) = F_x(x) \cdot F_y(y)$, $F_x$ being a Gaussian envelope and $F_y$ being the original Gabor filter as defined in Equation 6.7. Equation 6.14 is called the vertical 2-D Gabor filter.

We showed that it is possible to separate the 2-D Gabor filter in two components. It poses however a restriction on the orientation $\theta$ of the filter: only horizontal and vertical variants can be used. The set of separable Gabor filters we used in our experiments is shown in Figure 6.12.


(a) Even 2-D Gabor Filters

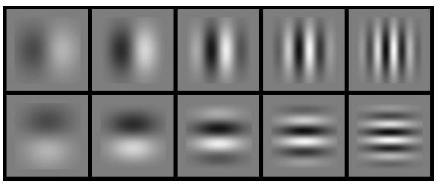

(b) Odd 2-D Gabor Filters

**Figure 6.12:** The horizontal and vertical 2-D Gabor filters. In (a) an ensemble of horizontal and vertical even 2-D Gabor filters are presented. In (b) the odd filters are presented. Five different frequencies are displayed, from left to right: $f$ = 0.5, 1, 2, 3 and 4. These values were used in our experiments.

### 2-D Combination Gabor Filters

We can only use the horizontal and vertical variants of the 2-D Gabor filter and this could pose a severe constraint on the ability to learn rotated features in the image. Therefore we created a 2-D combination filter that modulates the 1-D Gabor filter with another 1-D Gabor filter, rather than with a Gaussian. The original 2-D Gabor filter is rewritten into its 2-D combination form as follows:

$$\begin{aligned} G_c(x, y|\sigma'_x, \sigma'_y, f, \theta) &= e^{-\frac{1}{2}(\frac{x'^2}{\sigma'^2_x}+\frac{y'^2}{\sigma'^2_y})} \cdot e^{i2\pi f(x'+y')} \\ x' &= x\sin\theta + y\cos\theta \\ y' &= x\cos\theta - y\sin\theta \end{aligned} \tag{6.15}$$

Again only the horizontal and vertical variants can be used. This set of 2-D Gabor combination filters will produce diagonal squared filters that closely resemble to the diagonal Haar-like filter used by Viola and Jones (see Figure 6.13). Viola and Jones added the diagonal filter to their set of filters in order to be able to focus on diagonal parts of the image. By adding these 2-D Gabor combination filters to the set of separable 2-D Gabor filters, we hope to enrich the set with filters that are able to focus on diagonal aspects of the image.

### Computing Separable Gabor Filters On SIMD

Because the 2-D Gabor filters and 2-D Combination Gabor filters are separable, they can be easily computed on an SIMD architecture (see Section 5.3.2). However the cumulative row sum, which is used to speed up filter computation, cannot be used as the cumulative row sum requires binary separable filters (with coefficients -1 or 1).

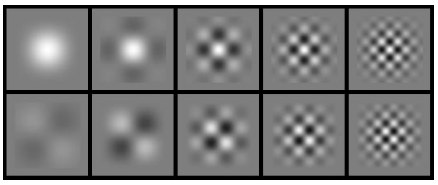


**Figure 6.13:** 2-D Combination Gabor Filters. We created a 2-D combination filter that modulates the 1-D Gabor filter with another 1-D Gabor filter in order to produce diagonal squared filters. The top row shows the even combination filters that are created by applying an even Gabor function on both the x and y-axes. The bottom row shows the odd combination filters which are created by applying an odd Gabor function on both axes. Note that the second filter on the bottom row closely resembles the diagonal Haar-like filter as used by Viola and Jones. Five different frequencies are displayed, from left to right: $f = 0.5$, 1, 2, 3 and 4. These values were used in our experiments.

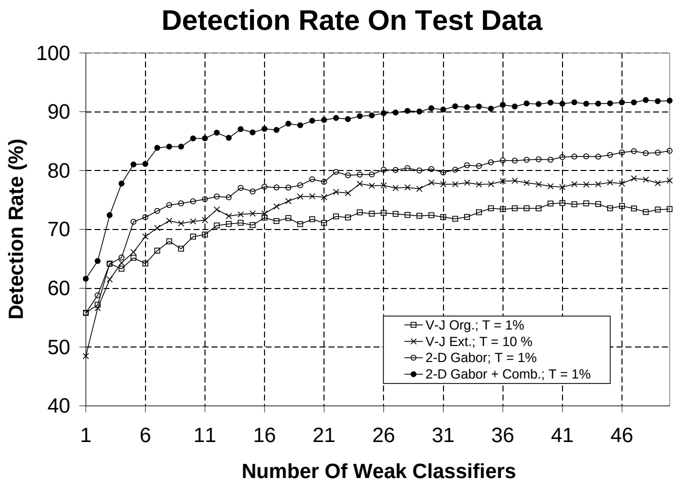
### 6.2.3 Experiments

To test the performance of the more complex filters we set up three experiments, each with a different filter set: the extended Viola and Jones set (Section 6.2.1), the horizontal and vertical 2-D Gabor filter set and the horizontal and vertical 2-D Gabor filter set enriched with the 2-D Combination Gabor filters. For the Gabor based sets we used five different frequency settings, $f = 0.5$, 1, 2, 3 and 4, and a fixed $\sigma$ value for the Gaussian envelope.

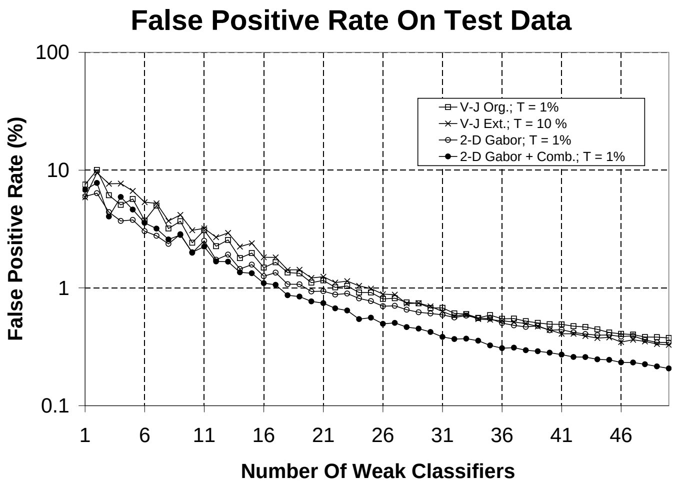
We trained 20 strong classifiers per experiment to create an average of the performance per filter set. The multiple threshold weak classifiers were used to train strong classifiers in each experiment. With a simple experiment, as described in Section 6.1, we determined the optimal value of $T$ per different filter set. Again we used the MIT+CBCL frontal face database for the training and test data.

The results of the experiments using the Extended Viola and Jones set (V-J Ext), the 2-D Gabor set and the 2-D Gabor set with combination filters (2-D Gabor + Comb.) are compared against the original Viola and Jones filter set (V-J Org) in the following section.

### 6.2.4 Results

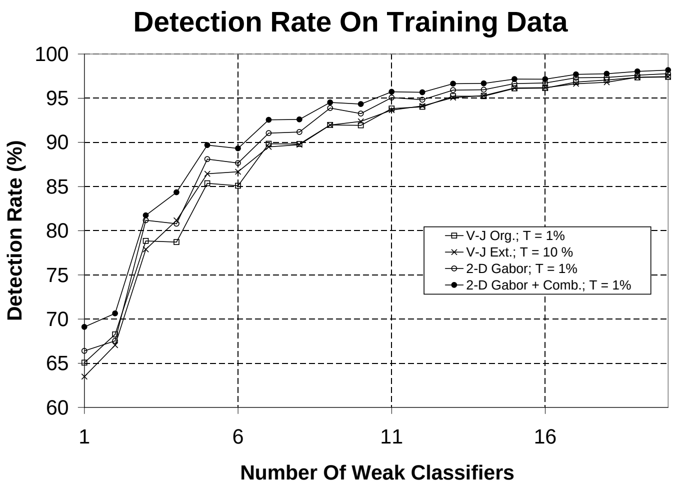


(a)

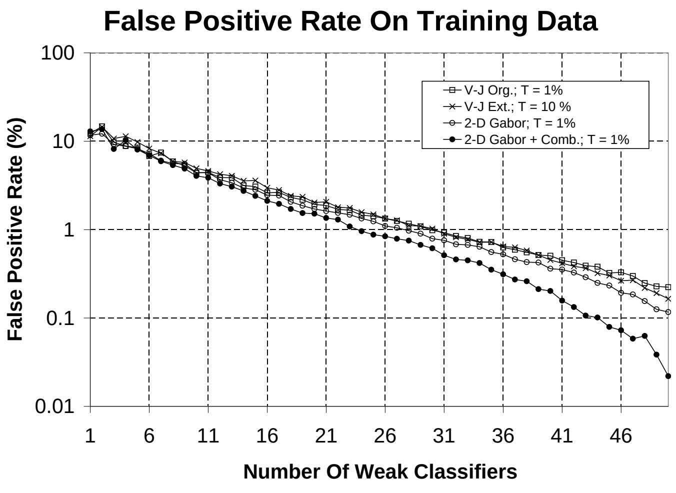


(b)

**Figure 6.14:** Detection rate and false positive rate on test data using weak classifiers with different filters sets. The detection rate of the complexer filter set is higher than when using the original filter set. Note that the 2-D Gabor without combination set, outperforms the original and extended Viola and Jones set even though it lacks diagonal filters. Clearly the 2-D Gabor set with combination filters outperforms all others. Also its false positive rate (shown in log domain) is much lower than the others.



(a)



(b)

**Figure 6.15:** Detection rate and false positive rate on training data using weak classifiers with different filters sets. The detection rate on the training data (only the first 20 classifiers are displayed) is near equal, although the 2-D Gabor with combination filters performed best. The false positive rate (shown in log domain) converges exponentially towards zero except for the 2-D Gabor set with combination filters: that converges even faster towards zero.

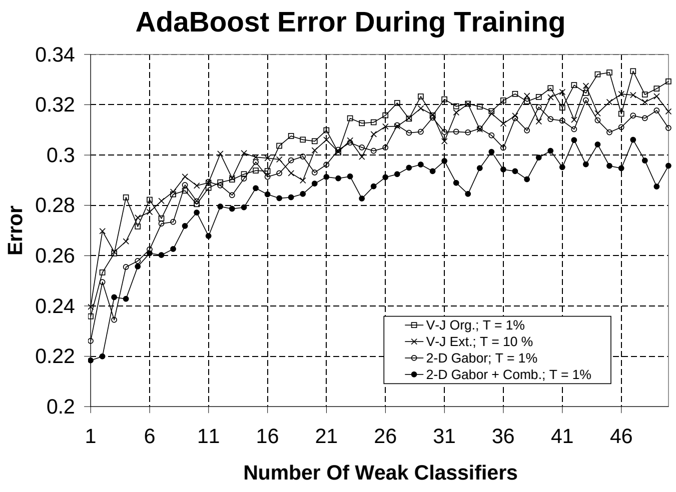


**Figure 6.16:** The AdaBoost error $\epsilon_t$ during training using single and multiple threshold weak classifiers. This figure shows again that the 2-D Gabor with combination filters outperforms all other filter sets.

The previous figures demonstrated better results in terms of detection rate and false positive rate, on both training and test set, when using the more complex filter sets were used. The 2-D Gabor with combination filters set performed best. To further compare the different filter sets, we determined an ROC curve of each strong classifier. The averaged ROC curves of the strong classifiers using different filter sets are displayed in Figure 6.17.

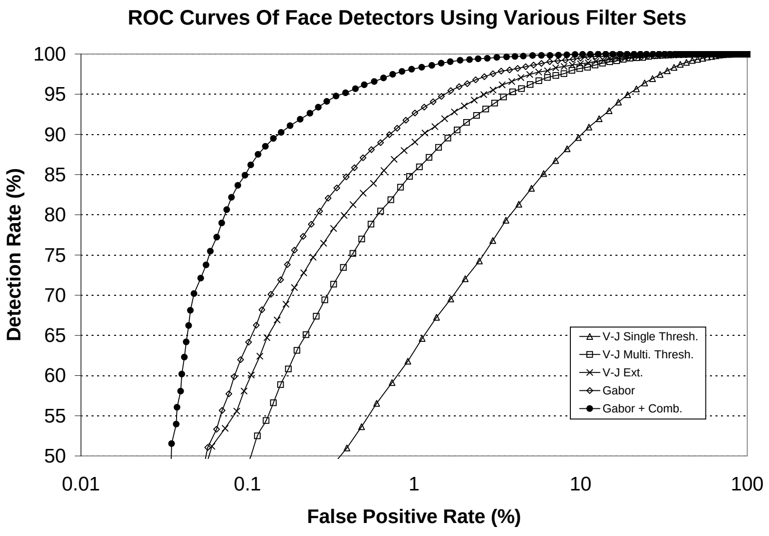


**Figure 6.17:** The averaged ROC curves of the strong classifiers using weak classifiers with different filters. The $x$-axis is presented in log domain, this better visualizes the differences between the various strong classifiers.

### 6.2.5 Discussion

In this section we proposed the usage of complexer filters for the weak classifiers. We showed that the performance in terms of detection rate and false positive rate is much better when using complexer filters. The filter set that performed best in our experiments was the 2-D Gabor set with 2-D combination filters. This filter set outperformed the others in terms of detection rate and false positive rate, both on training as well as test data. Using this filter set might reduce the total number of weak classifiers in the final strong classifier significantly. However, computing a separable Gabor filter on SIMD consumes more instructions than computing a Haar-like filter, this because the cumulative row sum cannot be used.

As already stressed in the discussion on multiple threshold classifiers, the results are preliminary and more research is required to obtain the final results.

# Chapter 7

# Conclusions

The primary objective of our research was to find a robust face detection technique which is able to run on the INCA$^+$. Using the face detection experiment described in Chapter 4, we concluded that the Viola and Jones detector outperforms the skin color based face detector. Furthermore we were able to implement a reasonable fast and accurate variant of the Viola and Jones detector on the INCA$^+$, which runs at 4-5 frames per second while maintaining a high performance.

The second objective was to investigate the possibility to implement the chosen face detector on the low-level Xetal processor. It proved to be impossible to implement the Viola and Jones detector on Xetal. However we presented a framework in Chapter 5 which enables the Viola and Jones face detector to run on an SIMD processor. We also introduced two filter computation methods which can be used in an SIMD environment to compute filters without using the integral image.

Finally, we wanted to improve the Viola and Jones algorithm in Chapter 6. We presented two improvements of the Viola and Jones algorithm. A general improvement that uses multiple threshold weak classifiers, proved to outperform the original single threshold weak classifiers. The second improvement uses two filter sets that are more complex than the Haar-like filter set used by Viola and Jones. We showed that when using these filters, the performance of the weak classifiers can improve significantly. These complex filters can be easily computed on an SIMD processor, using separable filters or the cumulative row sum.

## 7.1 Future Work

In this section we will present a direction of future research and possible improvements.

- **Further Investigate Proposed Improvements.** As already mentioned, the results of the proposed improvements are preliminary. More research is required in order to determine whether the proposed improvements perform significantly better.
- **Use Complex Filters For Detection Of Other Objects.** It would be an interesting research topic to determine how the complex filters perform on the detection of other objects (e.g. pedestrian detection, car detection etc.).
- **Use Complex Filters For Detection Of Profile Faces.** The original Viola and Jones filters proved to be too simple to be used for the detection of profile and rotated faces. A possible research topic would be to investigate whether the presented complex filters (or similar filters) perform better on the detection of profile and rotated faces.
- **Real Implementation On SIMD.** We presented a framework which enables the Viola and Jones face detector to run on an SIMD processor. Future research could focus on the actual implementation of the Viola and Jones detector on an SIMD processor.
- **Other Boosting Methods.** As mentioned in Section 3.5.10 other boosting methods, like FloatBoost and the *boosting chain*, perform better than the original Discrete AdaBoost. Using these boosting methods might reduce the number of weak classifiers significantly.

# Bibliography

[1] Open Source Computer Vision (OpenCV) Library. Intel Research, `http://www.intel.com/research/mrl/research/opencv/`.

[2] MIT CBCL Face Database #1. MIT Center For Biological and Computation Learning, `http://cbcl.mit.edu/cbcl/software-datasets/FaceData2.html`, 2000.

[3] *TriMedia Documentation: Book 1*, February 2003.

[4] A. Abbo and R. Kleihorst. *Xetal Software Framework Programming Guidelines.* Philips Research Laboratories, NatLab, 2001.

[5] G. Alaghband. Parallel computing and architectures. `http://carbon.cudenver.edu/~galaghba/csc6551.html`, 1997.

[6] D. G. Becker. Rhinoplasty. *Journal of Long-Term Effects of Medical Implants*, 13(3):223–246, 2003.

[7] R. N. Bracewell. *The Fourier Transform and Its Applications.*, chapter 3, page 27. McGraw-Hill, 3rd edition.

[8] D. Brown, I. Craw, and J. Lewthwaite. A SOM Based Approach to Skin Detection with Application in Real Time Systems. In *Proceedings of the British Machine Vision Conference*, pages 491–500, 2001.

[9] F. C. Crow. Summed-area tables for texture mapping. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 207–212. ACM Press, 1984.

[10] M. N. Dailey and G. W. Cottrell. Prosopagnosia in modular neural network models. In *Reggia, J., Ruppin, E., and Glanzman, D., Eds., Disorders of Brain, Behavior, and Cognition: The Neurocomputational Perspective, Progress in Brain Research series*, volume 121, pages 165–184. Elsevier Amsterdam, 1999.

[11] J. Daugman. Uncertainty relations for resolution in space, spatial frequency, and orientation optimized by two-dimensional visual cortical filters. *Journal of the Optical Society of America A*, 2:1160–1169, 1985.

[12] D. Dunn and W. Higgings. Optimal Gabor filters for texture segmentation. *IEEE Transactions on Image Processing*, 4(7):947–964, July 1995.

[13] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1998.

[14] D. Gabor. Theory of communications. In *Proc. Inst. Elect. Eng.*, volume 93, pages 429–459, 1946.

[15] C. Garcia and G. Tziritas. Face detection using quantized skin color regions merging and wavelet packet analysis. *IEEE TRANS. Multimedia*, 1(3):264–277, September 1999.

[16] E. Hjelmås and B. K. Low. Face detection: A survey. *Computer Vision and Image Understanding*, 83(3):236–274, 2001.

[17] M. Horton, P. Meenen, R. Adhami, and P. Cox. The Cost and Benefits of Using 2-D Gabor Filters in a Filter-Based Fingerprint-Matching System. In *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory*, pages 171–175, 18-19 March 2002.

[18] M. Jones and P. Viola. Fast multi-view face detection. Technical Report TR2003-96, Mitsubishi Electric Research Laboratories, June 2003.

[19] J. Kovac, P. Peer, and F. Solina. Human Skin Colour Clustering for Face Detection. *EUROCON 2003 - International Conference on Computer as a Tool*, September 2003.

[20] M. Lee, R. Kleihorst, A. Abbo, and E. Cohen-Solal. Real time skin-region detection with a single-chip digital camera. In *ICIP 2001*, Thessaloniki, Greece, Oct 8-11 2001.

[21] T. S. Lee. Image representation using 2D Gabor wavelets. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 18:959–971, 1996.

[22] S. Z. Li, Z. Q. Zhang, H. Y. Shum, and H. J. Zhang. FloatBoost Learning for Classification. In *Advances in Neural Information Processing Systems: Proceedings from the 2002 Conference*, pages 1017–1024, 2002.

[23] R. Lienhart, A. Kuranov, and V. Pisarevsky. Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection. *DAGM'03, 25th Pattern Recognition Symposium*, pages 297–304, September 2003.

[24] R. Lienhart and J. Maydt. An Extended Set of Haar-like Features for Rapid Object Detection. *IEEE ICIP 2002*, 1:900–903, September 2002.

[25] D. Maio and D. Maltoni. Real-Time Face Location on Gray-Scale Static Images. *Pattern Recognition*, 33(9):1525–1539, September 2000.

[26] R. McCready. Real-Time Face Detection on a Configurable Hardware System. In *Proceedings of Field-Programmable Logic and Applications, The Roadmap to Reconfigurable Computing, 10th International Workshop, FPL 2000*, volume 1896, pages 157–162, 2000.

[27] S. McKenna, S. Gong, and Y. Raja. Modelling facial colour and identity with gaussian mixtures. *Pattern Recognition*, 31(12):1883–1892, 1998.

[28] E. Osuna, R. Freund, and F. Girosi. Training support vector machines: an application to face detection. In *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, page 130. IEEE Computer Society, 1997.

[29] C. P. Papageorgiou, M. Oren, and T. Poggio. A General Framework for Object Detection. In *Proceedings of the Sixth International Conference on Computer Vision*, page 555. IEEE Computer Society, 1998.

[30] Perret et al. Visual cells in the temporal cortex sensitive to face view and gaze direction. In *Proc R Soc Lond B Biol Sci*, volume 223(1232), pages 293–317, Jan 22 1985.

[31] V. Philomin. Personal contact, Philips Aachen, 2003.

[32] D. Pollen and S. Ronner. Visual cortical neurons as localized spatial frequency filters. *IEEE Transactions on System, Man and Cybernetics*, 13:907–916, 1983.

[33] H. A. Rowley, S. Baluja, and T. Kanade. Neural network-based face detection. In *Proceedings of the 1996 Conference on Computer Vision and Pattern Recognition (CVPR '96)*, page 203. IEEE Computer Society, 1996.

[34] H. A. Rowley, S. Baluja, and T. Kanade. Rotation invariant neural network-based face detection. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, June 1998.

[35] E. Saber and A. M. Tekalp. Frontal-view face detection and facial feature extraction using color, shape and symmetry based cost functions. *Pattern Recognition Letters*, 19:669–680, 1998.

[36] G. Shakhnarovich, P. Viola, and B. Moghaddam. A Unified Learning Framework for Real-Time Face Detection and Classification. *IEEE International Conference on Automatic Face and Gesture Recognition (FG)*, pages 14–21, May 2002.

[37] M. C. Shin, K. I. Chang, and L. V. Tsap. Does Colorspace Transformation Make Any Difference on Skin Detection? *IEEE Workshop on Applications of Computer Vision*, pages 275–279, December 2002.

[38] S. K. Singh, D. S. Chauhan, M. Vatsa, and R. Singh. A Robust Skin Color Based Face Detection Algorithm. *Tamkang Journal of Science and Engineering*, 6(4):227–234, 2003.

[39] M. Störring, H. Andersen, and E. Granum. Skin colour detection under changing lighting condition. In *Araujo and J. Dias (ed.) 7th Symposium on Intelligent Robotics Systems*, pages 187–195, 1999.

[40] K. K. Sung and T. Poggio. Example-based learning for view-based human face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(1):39–51, 1998.

[41] K. K. Sung, T. Poggio, H. A. Rowley, S. Baluja, and T. Kanade. MIT+CMU Frontal Face Image Database. http://vasc.ri.cmu.edu/idb/html/face/frontal_images/index.html, 1997.

[42] J. W. Tanaka and M. J. Farah. Parts and wholes in face recognition. In *Quarterly Journal of Experimental Psychology*, volume 46A, pages 225–245., 1993.

[43] T. Theocharides, G. Link, N. Vijaykrishnan, M. J. Irwin, and W. Wolf. Embedded hardware face detection. In *Proceedings of the International Conference on VLSI Design, Mubai, India, January 2004*, pages 133–141. IEEE, January 2004.

[44] V. Vapnik and C. Cortes. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[45] V. Vezhnevets, V. Sazonov, and A. Andreeva. A Survey on Pixel-Based Skin Color Detection Techniques. In *Proc. Graphicon-2003*, pages 85–92, September 1998.

[46] P. Viola and M. Jones. Robust real-time object detection. Technical Report CRL 2001/01, The Cambridge Research Laboratory, February 2001.

[47] Y. Wang, C. Chua, and Y. Ho. Facial feature detection and face recognition from 2D and 3D images. *Pattern Recognition Letters*, 4(10):1191–1202, 2002.

[48] R. Xiao, L. Zhu, and H. J. Zhang. Boosting chain learning for object detection. In *Proceedings of the Ninth IEEE International Conference on Computer Vision*, pages 709–715, October 2003.

[49] J. Yang and A. Waibel. A real-time face tracker. In *Proc. Workshop on Applications of Computer Vision*, pages 142–147, 1996.

[50] M. H. Yang, D. J. Kriegman, and N. Ahuja. Detecting faces in images: A survey. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 24, January 2002.

[51] I. T. Young, L. J. van Vliet, and M. van Ginkel. Recursive gabor filtering. *IEEE Transactions On Signal Processing*, 50(11):2798–2805, 2002.

# Acknowledgments

I would like to thank the following people:

- My supervisors Ben Kröse and Richard Kleihorst for their guidance and help.
- Harry Broers for helping me with numerous programming problems and providing me with examples.
- Gerrianne Leeuwis for helping me labeling facial features of 800 different faces.
- Menno Bisschops for submitting my thesis.
- Collin van Ginkel for helping me with creating the cover of this thesis.
- Tom de Vries, Zbigniew Chamski and Paul Zander for helping me with various problems regarding makefiles and compilers.
- Anteneh Abbo for helping me with various Xetal related problems.
- Ivo Kronenburg for sharing his Xetal sources with me.
- Roel Reuvers for printing out this paper.

# Appendix A

# V-J On SIMD: A Performance Evaluation

This chapter presents a global indication of the performance required to run the Viola and Jones detector on an SIMD architecture. In Section A.1 we will present 12 models, covering 12 different SIMD configurations. These models can be used to compute an indication of the number of instructions required to compute the filters on an SIMD processor. Section A.2 will present an indication of the required memory when different SIMD configurations are considered.

This chapter assumes that the reader has sufficient knowledge about SIMD architectures and the Viola and Jones face detector.

## A.1 Required Number Of Instructions

In this thesis we presented two different SIMD memory models: fully- and limited connected memory models (see Section 2.1.1). Furthermore we presented three different filter computation methods for SIMD: filter computation using the integral image, using the cumulative row sum and using separable filters (see Section 5.2 and Section 5.3). For the detection of faces at multiple scales we can scale up the detection window, or use an image pyramid (see Section 5.5).

When the above settings are combined we can create 12 different configurations. For each configuration we created a model which can be used to compute the maximum[1]. The 12 models are presented in Table A.1. The abbreviations used in this table are explained in Table A.2.

[1] We present the maximum required number of instructions. For filters that are computed using the integral image, the maximum number of instructions is 9 (filter $F_{\text{diag}}$ requires 9 lookups as shown in Section 3.5.3). When using the cumulative row sum, the maximum is $w + 5$ (see Section 5.3.4), etc.

| Configuration | | | $ii$+$ii^2$ | $r$+$r^2$ | Filter Computation | Lighting Correction |
|---|---|---|---|---|---|---|
| FC | $ii$ | DS | $4 \times W$ | - | $9 \times S \times T \times \frac{W}{P}$ | $8 \times S \times \frac{W}{P}$ |
| FC | $r$ | DS | - | 4 | $\left(\sum_{i=0}^{S-1} ws^i + 5\right) \times T \times \frac{W}{P}$ | $\left(\sum_{i=0}^{S-1} ws^i + 2\right) \times 2 \times \frac{W}{P}$ |
| FC | SF | DS | - | - | $\left(\sum_{i=0}^{S-1} ws^i + h\right) \times T \times \frac{W}{P}$ | $\left(\sum_{i=0}^{S-1} ws^i + h\right) \times 2 \times \frac{W}{P}$ |
| LC | $ii$ | DS | $4 \times W$ | - | $\left(\sum_{i=0}^{S-1} \mathrm{MS}(i) + 9\right) \times T$ | $\left(\sum_{i=0}^{S-1} \mathrm{MS}(i) + 4\right) \times 2$ |
| LC | $r$ | DS | - | 4 | $\left(\sum_{i=0}^{S-1} ws^i + 5\right) \times T \times \frac{W}{P}$ | $\left(\sum_{i=0}^{S-1} ws^i + 2\right) \times 2 \times \frac{W}{P}$ |
| LC | SF | DS | - | - | $\left(\sum_{i=0}^{S-1} ws^i + h\right) \times T \times \frac{W}{P}$ | $\left(\sum_{i=0}^{S-1} ws^i + h\right) \times 2 \times \frac{W}{P}$ |
| FC | $ii$ | IP | $4 \times W$ | - | $9 \times T \times \frac{W}{P}$ | $8 \times \frac{W}{P}$ |
| FC | $r$ | IP | - | 4 | $(w + 5) \times T \times \frac{W}{P}$ | $(w + 2) \times 2 \times \frac{W}{P}$ |
| FC | SF | IP | - | - | $(w + h) \times T \times \frac{W}{P}$ | $(w + h) \times 2 \times \frac{W}{P}$ |
| LC | $ii$ | IP | $4 \times W$ | - | $(\mathrm{MS}(0) + 9) \times T$ | $(\mathrm{MS}(0) + 4) \times 2$ |
| LC | $r$ | IP | - | 4 | $(w + 5) \times T \times \frac{W}{P}$ | $(w + 2) \times 2 \times \frac{W}{P}$ |
| LC | SF | IP | - | - | $(w + h) \times T \times \frac{W}{P}$ | $(w + h) \times 2 \times \frac{W}{P}$ |

**Table A.1:** Viola and Jones detector on SIMD: maximum required instructions per line for filter computation, using 12 different configurations.

### A.1.1 Model Discussion

When looking at the 12 models we can observe that if the cumulative row sum ($r$) or separable filters (SF) are used, it does not matter if the used architecture is fully connected (FC) or limited connected (LC). We can explain this by looking at the way filters are computed when using separable filters or the cumulative row sum. To compute the filter result, both methods require a number of pixels to be summed. When summing a number of pixels, it does not matter whether the architecture is fully connected or limited connected because every pixel has to be addressed separately. Therefore when using separable filters or a cumulative row sum for filter computation, fully connected architectures will not perform better than limited connected architectures.

When using the integral image however, the fully connected architecture always outperforms the limited connected architecture. In a fully connected architecture, every integral image value can be accessed by every PE. In a limited connected architecture, only the integral image values of neighboring PEs are available for each PE. To reach other values, the PEs will have to shift (as explained in Section 2.1.1) and this consumes instructions.

When using an image pyramid the number of instructions is constant for all models, however the total execution time will be increased significantly. Normally the complete image is processed after $H$ lines (if $H$ is the image height). When using an image pyramid this number is much larger. Also computing an image pyramid requires additional computation time (or additional hardware). Image pyramids should only be used when the total number of instructions is too large when scaling up the detection window. For example, when using separable filters for filter computation, then the filter computation time depends on the width and height of the detection window. Scaling up the detection window, directly increases the computation time.

## A.2 Required Memory

The required memory depends on the chosen filter computation technique. When using the integral image for filter computation two framebuffers of memory are required. One for the integral image and one for the squared

| Abbr. | Full Name | Extra Description |
|---|---|---|
| FC | Fully Connected | Indicates the usage of a Fully Connected architecture, as explained in Section 2.1.1. |
| LC | Limited Connected | Indicates the usage of a Limited Connected architecture, as explained in Section 2.1.1. |
| $ii$ | Integral Image | Indicates the usage of the integral image for filter computation. $ii^2$ denotes the square integral image. |
| $r$ | Cumulative Row Sum | Indicates the usage of the cumulative row sum for filter computation. $r^2$ denotes the square cumulative row sum. |
| SF | Separable Filter | Indicates the usage of separable filters for filter computation. |
| DS | Detection Window Scales | Indicates the usage of the cumulative row sum for filter computation. |
| IP | Image Pyramid | Indicates the usage of an image pyramid. This ensures that filters can be computed in constant time. |
| $W$ | Image width | Denotes the image width in pixels. |
| $H$ | Image height | Denotes the image height in pixels. |
| $T$ | Number of weak classifiers | The total number of weak classifiers used in the strong classifier. |
| $S$ | Number of scales | The total number of different scales used for the detection window. |
| $P$ | Number of PEs | The total number of processing elements in the LPA of the SIMD processor. |
| $C$ | Connections per PE | The connectivity (in one direction) of each PE. E.g. if $C = 1$, the PE is connected to one neighbor in one direction. Xetal has $C = 1$. |
| $R$ | Range per PE | $R = \frac{W}{P} \times C$. This value indicates the number of pixels each PE can access of neighboring PEs without shifting. E.g. in case of Xetal: $W = 640$ pixels, $P = 320$ and $C = 1$, then $R = 2$. Which is true for Xetal since it can access 2 neighboring pixels in one direction. |
| MS($i$) | Maximum number of shifts at scale $i$ | This number represents the maximum number of shifts required to fetch the integral image values at scale $i$. This function returns the following value: $\text{MS}(i) = \sum_{j=0}^{\frac{W}{P}-1} \lceil \max(\frac{ws^i-(\frac{W}{P}-j)-R}{R}, 0) \rceil$. |
| $w$ | Org. det. window width | The width in pixels of the original detection window. E.g. 24 pixels. |
| $h$ | Org. det. window height | The height in pixels of the original detection window. E.g. 24 pixels. |
| $s$ | Scale factor | The scale factor of the detection window. E.g. 1.25. |

**Table A.2:** Abbreviations used in Table A.1.

integral image. The size of the memory elements in the framebuffer is: $\lceil {}^2\log_{W\times H\times 255}\rceil$ bits for the integral image framebuffer and $\lceil {}^2\log_{W\times H\times 255^2}\rceil$ bits for the squared integral image framebuffer. For instance when using a $640 \times 480$ image, the framebuffer containing the integral image is 480 lines. Each line contains 640 elements, each element is 27 bits in size. The framebuffer containing the squared integral image is 480 lines. Each line contains 640 elements, each element is 35 bits in size.

When using the cumulative row sum for filter computation also two framebuffers of memory are required. One for the cumulative row sum and for the squared cumulative row sum. The size of the memory elements in the framebuffer is: $\lceil {}^2\log_{H\times 255}\rceil$ bits for the integral image framebuffer and $\lceil {}^2\log_{H\times 255^2}\rceil$ bits for the squared integral image framebuffer. For instance when using a $640 \times 480$ image, the framebuffer containing the cumulative row sum is 480 lines. Each line contains 640 elements, each element is 17 bits in size. The framebuffer containing the squared cumulative row sum is 480 lines. Each line contains 640 elements, each element is 25 bits in size.

When using separable filters for filter computation, no additional framebuffer is required. This technique is computationally the most expensive, however it requires the least amount of memory.

# Appendix B

# Programming Details

## B.1 Programming Xetal

Table B.1 shows the Xetal Instruction Set for programming the LPA. Except for the `NOP` instruction, the result of an instruction is written to an optional line memory column (`res[i]`) and to the accumulator (`ACCU`). The value of index `i` is either 0 or 1 because each PE can only write to its own MU, which contains two columns: 0 and 1 (see also Figure 2.4).

Either the `ACCU`, a line memory column or a value from the GCP can serve as source operands for an instruction. The source operand designated as `oprd0` can be a line memory column (`lmem[i]`) or a value from the GCP, whereas `oprd1` can only be a line memory column. Since Xetal can read from its direct neighbors, the value of index `i` now ranges from -2 to 3. When `i` $= 0$ or 1, the data directly assigned to the PE is accessed. Data from a left neighbor is obtained by setting `i` $= -2$ or $-1$, and data from a right neighbor by setting `i` $= 2$ or 3.

While the accumulator is affected by all instructions, the flag is only affected by four compare instructions: `MAX, MIN, ABSMAX, ABSMIN`. This flag is used for the `PASSC` and `PASSCR` instructions. Xetal's LPA lacks conditional jump instructions like `if` since an `if` instruction can jump to two different locations depending on the outcome of a conditional check. Each location can contain different program code. Jumping to a different location in the LPA is thus prohibited by means of SIMD since each PE executes the same instructions. For this reason the Pass Conditional (`PASSC`) and its inverse, Pass Conditional Reverse (`PASSCR`), were designed. The `PASSC` is used in combination with the conditional flags. An example of how the `PASSC` and `PASSCR` instructions work as well as many other examples of how to program Xetal can be found in the next section.

| Instruction | Usage | Effect |
|---|---|---|
| NOP | NOP | *do nothing* |
| PASS | PASS res[i], oprd0; | res[i], ACCU $\leftarrow$ oprd0 |
| PASSC | PASSC res[i], oprd0; | res[i], ACCU $\leftarrow$ oprd0 $if\ flag = 1$ |
| | | res[i], ACCU $\leftarrow$ ACCU, $otherwise$ |
| PASSCR | PASSCR res[i], oprd0; | res[i], ACCU $\leftarrow$ oprd0 $if\ flag = 0$ |
| | | res[i], ACCU $\leftarrow$ ACCU, $otherwise$ |
| ADD | ADD res[i], oprd0; | res[i], ACCU $\leftarrow$ ACCU $+$ oprd0 |
| SUB | SUB res[i], oprd0; | res[i], ACCU $\leftarrow$ ACCU $-$ oprd0 |
| SUBA | SUBA res[i], oprd0; | res[i], ACCU $\leftarrow$ oprd0 $-$ ACCU |
| MUL | MUL res[i], oprd1, coef; | res[i], ACCU $\leftarrow$ oprd1 $*$ coef |
| MULA | MULA res[i], coef; | res[i], ACCU $\leftarrow$ ACCU $*$ coef |
| MAC | MAC res[i], oprd1, coef; | res[i], ACCU $\leftarrow$ ACCU $+$ (oprd1 $*$ coef) |
| ABS | ABS res[i], oprd0; | res[i], ACCU $\leftarrow$ \|oprd0\| |
| ABSA | ABSA res[i]; | res[i], ACCU $\leftarrow$ \|ACCU\| |
| MIN | MIN res[i], oprd0; | res[i], ACCU $\leftarrow$ $min$(ACCU, oprd0)<br>$flag = 1,\ if$ oprd0 $\leq$ ACCU<br>$flag = 0,\ otherwise$ |
| MAX | MAX res[i], oprd0; | res[i], ACCU $\leftarrow$ $max$(ACCU, oprd0)<br>$flag = 1,\ if$ oprd0 $>$ ACCU<br>$flag = 0,\ otherwise$ |
| ABSMIN | ABSMIN res[i], oprd0; | res[i], ACCU $\leftarrow$ oprd0, $if$ \|oprd0\| $\leq$ \|ACCU\|<br>res[i], ACCU $\leftarrow$ ACCU, $otherwise$<br>$flag = 1,\ if$ \|oprd0\| $\leq$ \|ACCU\|<br>$flag = 0,\ otherwise$ |
| ABSMAX | ABSMAX res[i], oprd0; | res[i], ACCU $\leftarrow$ oprd0, $if$ \|oprd0\| $>$ \|ACCU\|<br>res[i], ACCU $\leftarrow$ ACCU, $otherwise$<br>$flag = 1,\ if$ \|oprd0\| $>$ \|ACCU\|<br>$flag = 0,\ otherwise$ |

**Table B.1:** Xetal LPA Instruction Set [4].

### B.1.1 Xetal Programming Examples

This section presents some programming examples that demonstrate how Xetal works. Some examples mention the number of instructions that a particular program consumes. This number represents the number of instructions per pixel. Since each PE operates on two pixels (the even and odd columns) the total number of instructions would be twice as much usually, unless the even and odd pixels do not use the same algorithm.

### Pass Conditionals

In Figure B.1(a) an example of a simple program containing conditional jumps (`if`) is presented in pseudo code. This program could be easily implemented on a sequential processor and executes in 6 instructions. However this program uses the conditional jump which is prohibited on the Xetal's LPA, since it runs in SIMD mode. The Xetal equivalent code is presented in Figure B.1(b) and runs in 7 instructions. The example also demonstrates how successive `PASSC` or `PASSCR` can reuse the flag once it has been set.

```
my_value := 10
lmem_a[0] := 20
lmem_b[0] := max(my_value, lmem_a[0])

if (lmem_a[0] > my_value)
    lmem_c[0] := lmem_a[0]
    lmem_d[0] := my_value
else
    lmem_c[0] := my_value
    lmem_d[0] := lmem_a[0]
fi
```

(a) Program with conditional jumps

```
// Initialise the accumulator
ACCU := 10
// Init a line memory column
lmem_a[0] := 20

// Determine maximum
lmem_b[0] := MAX(ACCU, lmem_a[0])

// now lmem_b[0] := 20
// lmem_a[0] > ACCU -> flag is set

// ACCU is affected by all operations
// so re-initialise
ACCU := 10

// Do PASSC (flag is set)
lmem_c[0] := PASSC(ACCU, lmem_a[0])

// lmem_c[0] := lmem_a[0] := 20 since
// if is set PASSC passes oprd0

// ACCU is affected by all operations
// so re-initialise
ACCU := 10

// Do PASSCR (flag is still set)
lmem_d[0] := PASSCR(ACCU, lmem_a[0])

// lmem_d[0] := ACCU := 10 since
// if flag is set PASSCR passes ACCU
```

(b) Xetal LPA equivalent

**Figure B.1:** Example of the `PASSC` and `PASSCR` operations

## Simple Binary Threshold

Using the PASSC instruction it is easy to create a simple binary threshold. This example demonstrates how to implement such a threshold. The example also demonstrates how to program for the two columns (0 and 1): the code is simply copied.

Suppose a threshold is set at red value 50: all red pixels $> 50$ are set to 1 and all pixels $\leq 50$ are set to0.

```
thresh_value := 50

if (lmem_red[0] > thresh_value)
    lmem_red[0] := 1
else
    lmem_red[0] := 0
fi

if (lmem_red[1] > thresh_value)
    lmem_red[1] := 1
else
    lmem_red[1] := 0
fi
```

(a) Program with conditional jumps

```
//// Do column 0
// Load the red value
ACCU := lmem_red[0]

// Determine maximum
MAX(ACCU, 50)

// Load value on success
ACCU := 1

// Do PASSC, this will load 0 or 1
lmem_red[0] := PASSC(ACCU, 0)

// Load the red value
ACCU := lmem_red[1]

//// Do column 1
// Determine maximum
MAX(ACCU, 50)

// Load value on success
ACCU := 1

// Do PASSC, this will load 0 or 1
lmem_red[1] := PASSC(ACCU, 0)
```

(b) Xetal LPA equivalent

**Figure B.2:** Example of a simple binary threshold

## 5x1 Mean Filter

To implement a mean filter with width 5 pixels and height 1 pixel, Xetal's PE capability to read from its neighboring pixels needs to be used. A 5x1 mean filter is defined as:

$$mean(x, y) = \frac{\sum_{j=-2}^{2} I(x + j, y)}{5}$$

Where $mean(x, y)$ is the mean value at $(x, y)$ and $I$ is the actual image value. Xetal lacks the divide instruction, however every division can be rewritten into a multiplication with the reciprocal (multiplicative inverse) value:

$$\frac{a}{b} = a(\frac{1}{b}) \qquad a \in \Re, b \in \Re \backslash \{0\}$$

So instead we multiply each value with the reciprocal value 0.2. The resulting code is shown in Figure B.3.

```
lmem_a[0] += lmem_a[-2]
lmem_a[0] += lmem_a[-1]
lmem_a[0] += lmem_a[1]
lmem_a[0] += lmem_a[2]
lmem_a[0] = lmem_a[0] / 5

lmem_a[1] += lmem_a[-1]
lmem_a[1] += lmem_a[0]
lmem_a[1] += lmem_a[2]
lmem_a[1] += lmem_a[3]
lmem_a[1] = lmem_a[1] / 5
```

(a) Program with the divide instruction

```
//// Do column 0
ACCU := lmem_a[-2] * 0.2

// Use multiply accumulate
ACCU := ACCU + lmem_a[-1] * 0.2;
ACCU := ACCU + lmem_a[0] * 0.2;
ACCU := ACCU + lmem_a[1] * 0.2;
lmem_a[0] := ACCU +
                lmem_a[2] * 0.2;

//// Do column 1
ACCU := lmem_a[-1] * 0.2

// Use multiply accumulate
ACCU := ACCU + lmem_a[0] * 0.2;
ACCU := ACCU + lmem_a[1] * 0.2;
ACCU := ACCU + lmem_a[2] * 0.2;
lmem_a[1] = ACCU +
               lmem_a[3] * 0.2;
```

(b) Xetal LPA equivalent

**Figure B.3:** Example of a 5x1 mean filter

### Cumulative Row Sum

The cumulative row sum is defined as the sum of all pixels directly above a pixel:

$$rowsum(x, y) = \sum_{k=0}^{y} I(x, k)$$

Where $rowsum(x, y)$ is the cumulative row sum at point $(x, y)$. This can be rewritten into a recursive version [46]:

$$rowsum(x, y) = rowsum(x, y - 1) + I(x, y)$$

Where $rowsum(x, -1) = 0$. In Figure B.4 an example of a possible Xetal implementation of the cumulative row sum is shown. The example sums all rows of the red channel (`lmem_red`) and stores it into `lmem_rowsum`. The example also demonstrates the use of the GCP since it uses an `IF` statement. The `IF` statement is prohibited in LPA mode but the GCP can make use of it. In this example the GCP sends four instructions to the LPA when the register `current_line = 0` and six instructions to the LPA when `current_line != 0`. This way the SIMD principle is not violated since each processor still executes the same instructions at the same time.

```
// GCP IF
IF(current_line = 0) THEN
    lmem_rowsum[0] := lmem_red[0]
    lmem_prev[0] := lmem_rowsum[0]

    // ... same for column 1

    // 2 instructions on first line
ELSE
    // load previous value into ACCU
    ACCU := lmem_prev[0]

    // store rowsum
    lmem_rowsum[0] := ACCU + lmem_red[0]

    lmem_prev[0] := lmem_rowsum[0]

    // ... same for column 1

    // 3 instructions on other lines
FI
```

**Figure B.4:** Example of the cumulative row sum

### Row Shift

A *row shift* shifts all pixels of a row to the right or to the left.

$$shiftedrow[i](n) = row[i+n]$$

Where $shiftedrow$ is the shifted row, $n$ is the number of shifts and $0 \leq i \leq rowSize$. If $n$ is positive it shifts the row to the left, when it is negative the row is shifted to the right.

When shifting, some pixels will be shifted out and some will be shifted into the row. The question is only what to do with these pixels. Three possible solutions are:

**real shift** Truncate the pixels that are shifted out and assign value 0 to the pixels that are shifted in. For example (shift to the right 3 elements):

```
5 3 1 9 3 7 4 8    source
===============
0 0 0 5 3 1 9 3    real shift
```

**rotate** Rotate the row: every pixel that is shifted out shifts in at the other side of the row. This solution ensures that no data is lost. The rotate example:

```
5 3 1 9 3 7 4 8    source
===============
7 4 8 5 3 1 9 3    rotate
```

**Xetal shift** Truncate the pixels that are shifted out. When shifting to the right the value of the leftmost pixel will be assigned to the shifted in pixel. When shifting to the left the value of the rightmost pixel will be assigned. The Xetal shift example:

```
5 3 1 9 3 7 4 8    source
===============
5 5 5 5 3 1 9 3    Xetal shift
```

Xetal lacks a direct shift instruction and therefore it need to be implemented by copying values of neighboring PEs. Figure B.5 shows the pseudo source code of such an implementation. Because each PE of the Xetal's LPA is connected to two neighbors, as explained in Section 2.2, a single instruction (per pixel) can shift up to two columns.

```
//// single shift (1 column)
// right shift
linemem_a[1] := linemem_a[0]
linemem_a[0] := linemem_a[-1]

// left shift
linemem_a[0] := linemem_a[1]
linemem_a[1] := linemem_a[2]


//// double shift (2 columns)
// right shift
linemem_a[1] := linemem_a[-1]
linemem_a[0] := linemem_a[-2]

// left shift
linemem_a[0] := linemem_a[2]
linemem_a[1] := linemem_a[3]
```

**Figure B.5:** Example of the row shift

### Cumulative Column Sum

Instead of the cumulative row sum we can also define a cumulative column sum. The cumulative column sum is defined as the sum of all pixels left of a pixel:

$$colsum(x, y) = \sum_{k=0}^{x} I(k, y)$$

Where $colsum(x, y)$ is the cumulative column sum at point $(x, y)$. We can rewrite this into its recursive equivalent:

$$colsum(x, y) = colsum(x - 1, y) + I(x, y)$$

Where $colsum(-1,\ y) = 0$. However this introduces a problem: $colsum(x,\ y)$ depends on the cumulative column sum of the previous pixel: $colsum(x - 1,\ y)$. But since Xetal's LPA operates in SIMD mode that value will be computed at exactly the same time as $colsum(x,\ y)$ itself and is therefore available only after the computation. Because of this problem the simple recursive equivalent is not possible on Xetal and calculating the cumulative column sum can only be done by summing all pixels individually as shown in Figure B.6.

This example uses the row shift principle as explained in the previous example and combines it with an addition each iteration. It starts with the original image value (in this case `lmem_red`) and then adds the left neighboring pixel to that value. Finally it shifts the complete row to the right and subsequently adds the left neighboring pixel again, etc. When shifting the complete row to the right the rightmost pixel value will be

```
colsum[0] := lmem_red[0]
colsum[1] := lmem_red[1]

// GCP FOR LOOP
FOR i := 0 TO 319
    ACCU := colsum[0]

    // add left neighbor
    colsum[0] = ACCU + lmem_red[-1]

    ACCU := colsum[1]

    // add left neighbor
    colsum[1] := ACCU + lmem_red[0]

    // shift lmem_red to the right
    lmem_red[1] := lmem_red[0]
    lmem_red[0] := lmem_red[-1]
LOOP

// 319*3 + 1 = 958 instr. per pixel
// total 1921*2 = 1916 instructions
```

**Figure B.6:** Example of the cumulative column sum

truncated and a new pixel value appears at the leftmost end of the row. Because of the fact that when shifting to the right the leftmost pixel value propagates to the right (see Section B.1.1) the leftmost pixel value should be 0. Otherwise it keeps adding the leftmost pixel value to the results each iteration.

After 320 iterations the algorithm is finally finished, consuming several hundreds of instructions. The algorithm could be made faster using the double shift technique as explained in Section B.1.1. This could save a number of instructions, but it would still consume a large amount of instructions.

Compared to the cumulative row sum, the cumulative column sum is far more computationally expensive and because of the amount of instructions even impossible on Xetal.

## B.2 Programming TriMedia

Programming the TriMedia processor can be done using the Rhapsody C/C++ software package. This package contains several functions which relief the programmer from doing low-level programming. The C/C++ code should be compiled with the TriMedia Compilation System (TCS) 2.2 tools. Also a small operating system called pSOS 2.2 that runs on the INCA$^+$ should be linked with the project. This OS is required for programs to run on the INCA$^+$. pSOS 2.2 is shipped with TCS 2.2.