Drowsy Driver Detection System

Design Project

By: Neeta Parmar
Instructor: Peter Hiscocks

Department of Electrical and Computer Engineering,
Ryerson University.

© 2002. All Rights Reserved.

CERTIFICATE OF AUTHORSHIP

I, Neeta Parmar, hereby declare that the work presented in this Engineering Design Project is my own to the best of my abilities, skills and knowledge except otherwise noted. Any instance omissions where credit has not been given are purely due to oversight and not an attempt to claim the work as my own.

______________________________ ____________________________

Signature of Author ***Date***

ACKNOWLEDGMENTS

First and foremost, I would like to thank God for blessing me with the strength, intelligence, and patience to complete this project. I would also like to thank my family for their continuing love and support, especially my father, who has been an extraordinary mentor. Thank you Peter Hiscocks for your advice throughout the project. And finally, I would like to thank Syed Quadri, Kim Woon-Fat, Stephen Eagle and Sangita Patel for allowing me to use their pictures in this report.

Contents

Abstract 6

List of Figures 7

1 Introduction 8

2 System Requirements 9

3 Report Organization 10

4 Literature Review 11

4.1 Techniques for Detecting Drowsy Drivers 11
4.1.1 Monitoring Physiological Characteristics 11
4.1.2 Other Methods 11

5 Design Issues 13

5.1 Illumination 13
5.2 Camera Hardware 14
5.3 Frame Grabbers and Capture Hardware 14
5.3.1 Homemade Frame Grabbers Using the Parallel Port 14
5.3.2 Commercial Frame Grabbers: Euresys Picolo I 15

6 Design 16

6.1 Concept Design 16
6.2 System Design 17
6.2.1 Background and Ambient Light 17
6.2.2 Camera 17
6.2.3 Light Source 17

7 Algorithm Development 19

7.1 System Flowchart 19
7.2 System Process 19
7.2.1 Eye Detection Function 19
Binarization 21
Face Top and Width Detection 22
Removal of Noise 25
Finding Intensity Changes on the Face 26
Detection of Vertical Eye Position 27

8 Algorithm Implementation **30**

8.1 Real-time System 30
8.2 Challenges 31
8.2.1 Obtaining the image 31
8.2.2 Constructing an effective light source 31
8.2.3 Determining the correct binarization threshold 31
8.2.4 Not being able to use edge detection algorithms 31
8.2.5 Case when the driver's head is titled 32
8.2.6 Finding the top of the head correctly 32
8.2.7 Finding bounds of the functions 33

9 Results and Future Work **34**

9.1 Simulation Results 34
9.2 Limitations 36
9.3 Future Work 37

10 Conclusion **38**

Appendix A **39**

Program Listing 39

Bibliography **61**

Abstract

A Drowsy Driver Detection System has been developed, using a non-intrusive machine vision based concepts. The system uses a small monochrome security camera that points directly towards the driver's face and monitors the driver's eyes in order to detect fatigue. In such a case when fatigue is detected, a warning signal is issued to alert the driver. This report describes how to find the eyes, and also how to determine if the eyes are open or closed. The algorithm developed is unique to any currently published papers, which was a primary objective of the project. The system deals with using information obtained for the binary version of the image to find the edges of the face, which narrows the area of where the eyes may exist. Once the face area is found, the eyes are found by computing the horizontal averages in the area. Taking into account the knowledge that eye regions in the face present great intensity changes, the eyes are located by finding the significant intensity changes in the face. Once the eyes are located, measuring the distances between the intensity changes in the eye area determine whether the eyes are open or closed. A large distance corresponds to eye closure. If the eyes are found closed for 5 consecutive frames, the system draws the conclusion that the driver is falling asleep and issues a warning signal. The system is also able to detect when the eyes cannot be found, and works under reasonable lighting conditions.

List of Figures

FIGURE 6.1: CASE WHERE NO RETINAL REFLECTION PRESENT. 16
FIGURE 6.2: PHOTOGRAPH OF DROWSY DRIVER DETECTION SYSTEM PROTOTYPE. 18
FIGURE 7.1: FLOW CHART OF SYSTEM. 20
FIGURE 7.2: EXAMPLES OF BINARIZATION USING DIFFERENT THRESHOLDS. 21
FIGURE 7.3: FACE TOP AND WIDTH DETECTION. 23
FIGURE 7.4: FACE EDGES FOUND AFTER FIRST TRIAL. 24
FIGURE 7.5: BINARY PICTURE AFTER NOISE REMOVAL. 25
FIGURE 7.6: FACE EDGES FOUND AFTER SECOND TRIAL. 25
FIGURE 7.7: LABELS OF TOP OF HEAD, AND FIRST TWO INTENSITY CHANGES. 26
FIGURE 7.8: RESULT OF USING HORIZONTAL AVERAGES TO FIND VERTICAL POSITION OF THE EYE. 27
FIGURE 7.9: COMPARISON OF OPEN AND CLOSED EYE. 28
FIGURE 8.1: RESULTS OF USING SOBEL EDGE DETECTION. 32
FIGURE 9.1: DEMONSTRATION OF FIRST STEP IN THE PROCESS FOR FINDING THE EYES – ORIGINAL IMAGE. 34
FIGURE 9.2: DEMONSTRATION OF SECOND STEP – BINARY IMAGE. 35
FIGURE 9.3: DEMONSTRATION OF THIRD STEP – INITIAL EDGE DETECTION IMAGE. 35
FIGURE 9.4: DEMONSTRATION OF THE FOURTH STEP - REMOVING BLOBS IN BINARY IMAGE. 35
FIGURE 9.5: DEMONSTRATION OF STEP 5 - SECOND EDGE DETECTION. 36
FIGURE 9.6: RESULT OF FINAL STEP - FINDING THE LEFT EYE USING INTENSITY INFORMATION. 36

Chapter 1

Introduction

Driver fatigue is a significant factor in a large number of vehicle accidents. Recent statistics estimate that annually 1,200 deaths and 76,000 injuries can be attributed to fatigue related crashes [9].

The development of technologies for detecting or preventing drowsiness at the wheel is a major challenge in the field of accident avoidance systems. Because of the hazard that drowsiness presents on the road, methods need to be developed for counteracting its affects.

The aim of this project is to develop a prototype drowsiness detection system. The focus will be placed on designing a system that will accurately monitor the open or closed state of the driver's eyes in real-time.

By monitoring the eyes, it is believed that the symptoms of driver fatigue can be detected early enough to avoid a car accident. Detection of fatigue involves a sequence of images of a face, and the observation of eye movements and blink patterns.

The analysis of face images is a popular research area with applications such as face recognition, virtual tools, and human identification security systems. This project is focused on the localization of the eyes, which involves looking at the entire image of the face, and determining the position of the eyes by a self developed image-processing algorithm. Once the position of the eyes is located, the system is designed to determine whether the eyes are opened or closed, and detect fatigue.

Chapter 2

System Requirements

The requirements for an effective drowsy driver detection system are as follows:

- A non-intrusive monitoring system that will not distract the driver.
- A real-time monitoring system, to insure accuracy in detecting drowsiness.
- A system that will work in both daytime and nighttime conditions.

The above requirements are subsequently the aims of this project. The project will consist of a concept level system that will meet all the above requirements.

Chapter 3

Report Organization

The documentation for this project consists of 10 chapters. Chapter 4 represents the Literature Review, which serves as an introduction to current research on driver drowsiness detection systems. Chapter 5 discusses the design issues of the project, specifically, the issues and concepts behind real-time image processing. Following this, Chapter 6 describes the design of the system, and chapter 7 describes the algorithm behind the system. Chapter 8 gives additional information of the real-time system and the challenges met. Chapter 9 shows images illustrating the steps taken in localizing the eyes, which is followed by a discussion on some possible future directions for the project, after which the final conclusions are drawn in chapter 10.

Chapter 4

Literature Review

4.1 Techniques for Detecting Drowsy Drivers

Possible techniques for detecting drowsiness in drivers can be generally divided into the following categories: sensing of physiological characteristics, sensing of driver operation, sensing of vehicle response, monitoring the response of driver.

4.1.1 Monitoring Physiological Characteristics

Among these methods, the techniques that are best, based on accuracy are the ones based on human physiological phenomena [9]. This technique is implemented in two ways: measuring changes in physiological signals, such as brain waves, heart rate, and eye blinking; and measuring physical changes such as sagging posture, leaning of the driver's head and the open/closed states of the eyes [9]. The first technique, while most accurate, is not realistic, since sensing electrodes would have to be attached directly onto the driver's body, and hence be annoying and distracting to the driver. In addition, long time driving would result in perspiration on the sensors, diminishing their ability to monitor accurately. The second technique is well suited for real world driving conditions since it can be non-intrusive by using optical sensors of video cameras to detect changes.

4.1.2 Other Methods

Driver operation and vehicle behaviour can be implemented by monitoring the steering wheel movement, accelerator or brake patterns, vehicle speed, lateral acceleration, and lateral displacement. These too are non-intrusive ways of detecting drowsiness, but are limited to vehicle type and driver conditions. The final technique for detecting drowsiness is by

monitoring the response of the driver. This involves periodically requesting the driver to send a response to the system to indicate alertness. The problem with this technique is that it will eventually become tiresome and annoying to the driver.

Chapter 5

Design Issues

The most important aspect of implementing a machine vision system is the image acquisition. Any deficiencies in the acquired images can cause problems with image analysis and interpretation. Examples of such problems are a lack of detail due to insufficient contrast or poor positioning of the camera: this can cause the objects to be unrecognizable, so the purpose of vision cannot be fulfilled.

5.1 Illumination

A correct illumination scheme is a crucial part of insuring that the image has the correct amount of contrast to allow to correctly process the image. In case of the drowsy driver detection system, the light source is placed in such a way that the maximum light being reflected back is from the face. The driver's face will be illuminated using a 60W light source. To prevent the light source from distracting the driver, an 850nm filter is placed over the source. Since 850nm falls in the infrared region, the illumination cannot be detected by the human eye, and hence does not agitate the driver. Since the algorithm behind the eye monitoring system is highly dependant on light, the following important illumination factors to consider are [1]:

1. Different parts of objects are lit differently, because of variations in the angle of incidence, and hence have different brightness as seen by the camera.
2. Brightness values vary due to the degree of reflectivness of the object.
3. Parts of the background and surrounding objects are in shadow, and can also affect the brightness values in different regions of the object.
4. Surrounding light sources (such as daylight) can diminish the effect of the light source on the object.

5.2 Camera Hardware

The next item to be considered in image acquisition is the video camera. Review of several journal articles reveals that face monitoring systems use an infrared-sensitive camera to generate the eye images [3],[5],[6],[7]. This is due to the infrared light source used to illuminate the driver's face. CCD cameras have a spectral range of 400-1000nm, and peak at approximately 800nm. The camera used in this system is a Sony CCD black and white camera. CCD camera digitize the image from the outset, although in one respect – that signal amplitude represents light intensity – the image is still analog.

5.3 Frame Grabbers and Capture Hardware

The next stage of any image acquisition system must convert the video signal into a format, which can be processed by a computer. The common solution is a frame grabber board that attaches to a computer and provides the complete video signal to the computer. The resulting data is an array of greyscale values, and may then be analysed by a processor to extract the required features. Two options were investigated when choosing the system's capture hardware. The first option is designing a homemade frame grabber, and the second option is purchasing a commercial frame grabber. The two options are discussed below.

5.3.1 Homemade Frame Grabbers: Using the Parallel Port

Initially, a homemade frame grabber was going to be used. The design used was based on the 'Dirt Cheap Frame Grabber (DCFG)', developed by Michael Day [2].

A detailed circuit description of the DCFG is beyond the scope of this report, but important observations of this design were made. The DCFG assumes that the video signal will be NTSC type compiling to the RS170 video standard. Although the DCFG successfully grabs the video signal and digitizes it, the limitation to this design was that the parallel port could not transfer the signal fast enough for real-time purposes. The DCFG was tested on two different computers, a PC (Pentium I – 233MHz) and a laptop (Pentium III – 700MHz). The laptop parallel port was much slower than the PC's. Using the PC, which has a parallel port of 4MHz, it was calculated that it would take approximately a minute to transfer a 480 x 640 pixel image. Because of the slow transfer rate of the parallel port it was concluded that this type of frame grabber could not be used.

Another option was to build a frame grabber similar to the commercial versions. This option was not pursued, since the main objective of the project is not to build a frame grabber. Building a complete frame grabber is a thesis project in itself, and since the parallel port design could not be used for real-time applications, the final option was purchasing a commercial frame grabber.

5.3.2 Commercial Frame Grabbers: Euresys Picolo I

The commercial frame grabber chosen for this system is the Euresys Picolo I. The Picolo frame grabber acquired both colour and monochrome image formats. The Picolo supports the acquisition and the real-time transfer of full resolution colour images (up to 768 x 576 pixels) and sequences of images to the PC memory (video capture). The board supports PCI bus mastering; images are transferred to the PC memory using DMA in parallel with the acquisition and processing. With the commercial board, it is also easier to write software to process the images for each systems own application. The board comes with various drivers and libraries for different software (i.e.; Borland C, Java, etc). The drivers allow the controlling of capturing and processing of images via custom code.

Chapter 6

Design

This chapter aims to present my design of the Drowsy Driver Detection System. Each design decision will be presented and rationalized, and sufficient detail will be given to allow the reader to examine each element in its entirety.

6.1 Concept Design

As seen in the various references [3],[5],[6],[7],[8],[9], there are several different algorithms and methods for eye tracking, and monitoring. Most of them in some way relate to features of the eye (typically reflections from the eye) within a video image of the driver.

The original aim of this project was to use the retinal reflection (only) as a means to finding the eyes on the face, and then using the absence of this reflection as a way of detecting when the eyes are closed. It was then found that this method might not be the best method of monitoring the eyes fort two reasons. First, in lower lighting conditions, the amount of retinal reflection decreases; and second, if the person has small eyes the reflection may not show, as seen below in Figure 6.1.

Figure 6.1: Case where no retinal reflection present.

As the project progressed, the basis of the horizontal intensity changes idea from paper [7] was used. One similarity among all faces is that eyebrows are significantly different from the skin in intensity, and that the next significant change in intensity, in the y-direction, is the eyes. This facial characteristic is the centre of finding the eyes on the face, which will allow the system to monitor the eyes and detect long periods of eye closure.

Each of the following sections describes the design of the drowsy driver detection system.

6.2 System Configuration

6.2.1 Background and Ambient Light

Because the eye tracking system is based on intensity changes on the face, it is crucial that the background does not contain any object with strong intensity changes. Highly reflective object behind the driver, can be picked up by the camera, and be consequently mistaken as the eyes. Since this design is a prototype, a controlled lighting area was set up for testing. Low surrounding light (ambient light) is also important, since the only significant light illuminating the face should come from the drowsy driver system. If there is a lot of ambient light, the effect of the light source diminishes. The testing area included a black background, and low ambient light (in this case, the ceiling light was physically high, and hence had low illumination). This setup is somewhat realistic since inside a vehicle, there is no direct light, and the background is fairly uniform.

6.2.2 Camera

The drowsy driver detection system consists of a CCD camera that takes images of the driver’s face. This type of drowsiness detection system is based on the use of image processing technology that will be able to accommodate individual driver differences. The camera is placed in front of the driver, approximately 30 cm away from the face. The camera must be positioned such that the following criteria are met:

1. The driver’s face takes up the majority of the image.
2. The driver’s face is approximately in the centre of the image.

The facial image data is in 480x640 pixel format and is stored as an array through the predefined Picolo driver functions (as described in a later section).

6.2.3 Light Source

For conditions when ambient light is poor (night time), a light source must be present to compensate. Initially, the construction of an infrared light source using infrared LED was going to be implemented. It was later found that at least 50 LEDs would be needed so create a source that would be able to illuminate the entire face. To cut down cost, a simple desk light was used. Using the desk light alone could not work, since the bright light is blinding if

looked at directly, and could not be used to illuminate the face. However, light from light bulbs and even daylight all contain infrared light; using this fact, it was decided that if an infrared filter was placed over the desk lamp, this would protect the eyes from a strong and distracting light and provide strong enough light to illuminate the face. A wideband infrared filter was placed over the desk lamp, and provides an excellent method of illuminating the face. The spectral plot of the filter is shown in Appendix A.

The prototype of the Drowsy Driver Detection System is shown in Figure 6.2.

Figure 6.2: Photograph of Drowsy Driver Detection System prototype

Chapter 7

Algorithm Development

7.1 System Flowchart

A flowchart of the major functions of The Drowsy Driver Detection System is shown in Figure 7.1.

7.2 System Process

7.2.1 Eye Detection Function

An explanation is given here of the eye detection procedure.

After inputting a facial image, pre-processing is first performed by binarizing the image.

The top and sides of the face are detected to narrow down the area of where the eyes exist. Using the sides of the face, the centre of the face is found, which will be used as a reference when comparing the left and right eyes.

Moving down from the top of the face, horizontal averages (average intensity value for each y coordinate) of the face area are calculated. Large changes in the averages are used to define the eye area.

The following explains the eye detection procedure in the order of the processing operations. All images were generating in Matlab using the image processing toolbox.

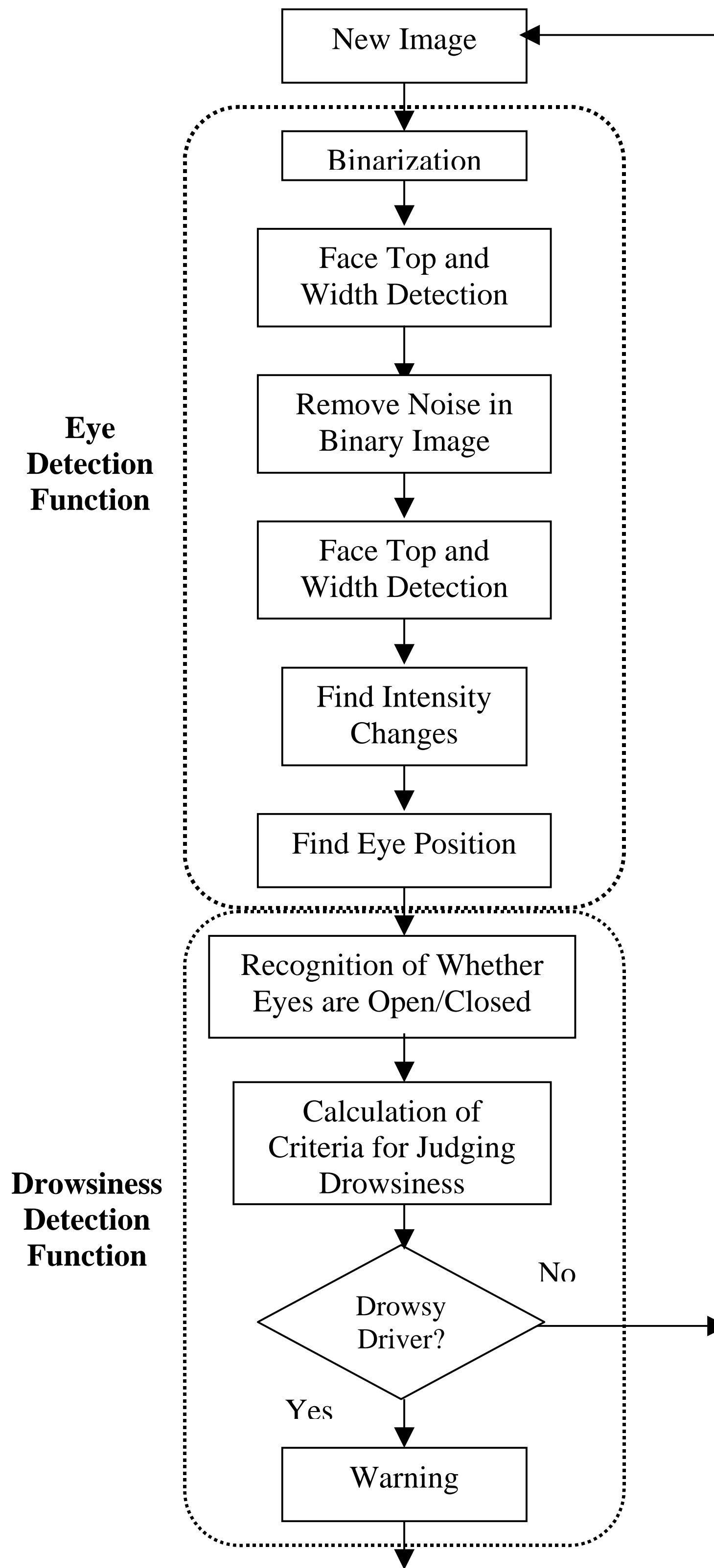


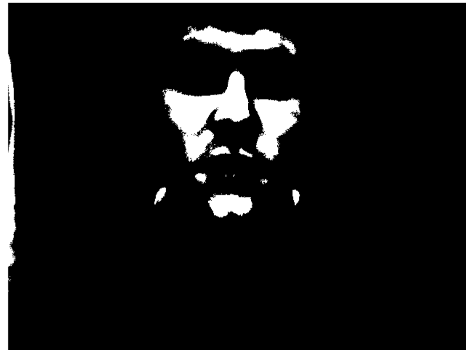
Figure 7.1: Flow chart of system.

Binarization

The first step to localize the eyes is binarizing the picture. Binarization is converting the image to a binary image. Examples of binarized images are shown in Figure 7.2.

a) Threshold value 100.

b) Threshold value 150.



c) Threshold value 200.

Figure 7.2: Examples of binarization using different thresholds.

A binary image is an image in which each pixel assumes the value of only two discrete values. In this case the values are 0 and 1, 0 representing black and 1 representing white. With the binary image it is easy to distinguish objects from the background. The greyscale image is converting to a binary image via thresholding. The output binary image has values of 0 (black) for all pixels in the original image with luminance less than level and 1 (white) for all other pixels. Thresholds are often determined based on surrounding lighting conditions, and the complexion of the driver. After observing many images of different faces under various lighting conditions a threshold value of 150 was found to be effective. The criteria used in choosing the correct threshold was based on the idea that the binary image of the driver's face should be majority white, allowing a few black blobs from the eyes, nose and/or lips. Figure 7.2 demonstrates the effectiveness of varying threshold values. Figure

7.2a, 7.2b, and 7.2c use the threshold values 100, 150 and 200, respectively. Figure 7.2b is an example of an optimum binary image for the eye detection algorithm in that the background is uniformly black, and the face is primary white. This will allow finding the edges of the face, as described in the next section.

Face Top and Width Detection

The next step in the eye detection function is determining the top and side of the driver's face. This is important since finding the outline of the face narrows down the region in which the eyes are, which makes it easier (computationally) to localize the position of the eyes. The first step is to find the top of the face. The first step is to find a starting point on the face, followed by decrementing the y-coordinates until the top of the face is detected. Assuming that the person's face is approximately in the centre of the image, the initial starting point used is (100,240). The starting x-coordinate of 100 was chosen, to insure that the starting point is a black pixel (no on the face). The following algorithm describes how to find the actual starting point on the face, which will be used to find the top of the face.

1. Starting at (100,240), increment the x-coordinate until a white pixel is found. This is considered the left side of the face.

2. If the initial white pixel is followed by 25 more white pixels, keep incrementing x until a black pixel is found.

3. Count the number of black pixels followed by the pixel found in step2, if a series of 25 black pixels are found, this is the right side.

4. The new starting x-coordinate value (x1) is the middle point of the left side and right side.

Figure 7.3 demonstrates the above algorithm.

Using the new starting point (x1, 240), the top of the head can be found. The following is the algorithm to find the top of the head:

1. Beginning at the starting point, decrement the y-coordinate (i.e.; moving up the face).

2. Continue to decrement y until a black pixel is found. If y becomes 0 (reached the top of the image), set this to the top of the head.

3. Check to see if any white pixels follow the black pixel.

   i. If a significant number of white pixels are found, continue to decrement y.
   ii. If no white pixels are found, the top of the head is found at the point of the initial black pixel.

☆ Starting point (100,240)
A Count 25 white pixels from this point
B Count 25 black pixels from this point
x1 Middle point of A and B

Figure 7.3: Face top and width detection.

Once the top of the driver's head is found, the sides of the face can also be found. Below are the steps used to find the left and right sides of the face.

1. Increment the y-coordinate of the top (found above) by 10. Label this y1 = y + top.

2. Find the centre of the face using the following steps:

   i. At point (x1, y1), move left until 25 consecutive black pixels are found, this is the left side (lx).
   ii. At point (x1, y1), move right until 25 consecutive white pixels are found, this is the right side (rx).
   iii. The centre of the face (in x-direction) is: (rx – lx)/2. Label this x2.

3. Starting at the point (x2, y1), find the top of the face again. This will result in a new y-coordinate, y2.

4. Finally, the edges of the face can be found using the point (x2, y2).

    i. Increment y-coordinate.
    ii. Move left by decrementing the x-coordinate, when 5 black consecutive pixels are found, this is the left side, add the x-coordinate to an array labelled 'left_x'.
    iii. Move right by incrementing the x-coordinate, when 5 black consecutive pixels are found, this is the right side, add the x-coordinate to an array labelled 'right_x'.
    iv. Repeat the above steps 200 times (200 different y-coordinates).

The result of the face top and width detection is shown in Figure 7.4, these were marked on the picture as part of the computer simulation.

Figure 7.4: Face edges found after first trial.

As seen in Figure 7.4, the edges of the face are not accurate. Using the edges found in this initial step would never localize the eyes, since the eyes fall outside the determined boundary of the face. This is due to the blobs of black pixels on the face, primarily in the eye area, as seen in Figure 7.2b. To fix this problem, an algorithm to remove the black blobs was developed.

Removal of Noise

The removal of noise in the binary image is very straightforward. Starting at the top, (x2,y2), move left on pixel by decrementing x2, and set each y value to white (for 200 y values). Repeat the same for the right side of the face. The key to this is to stop at left and right edge of the face; otherwise the information of where the edges of the face are will be lost. Figure 7.5, shows the binary image after this process.

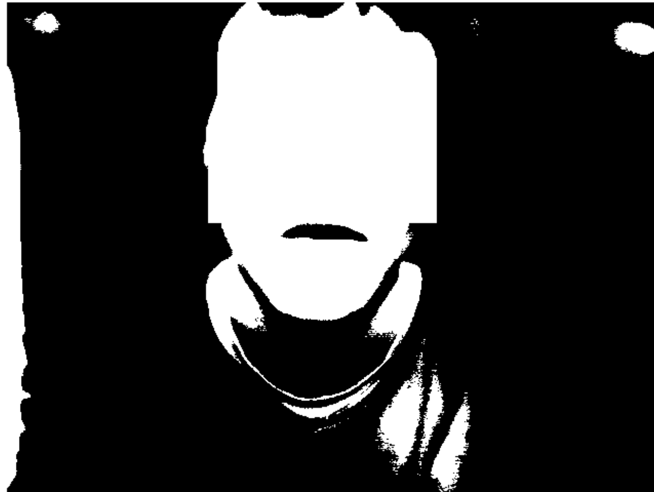


Figure 7.5: Binary picture after noise removal.

After removing the black blobs on the face, the edges of the face are found again. As seen below, the second time of doing this results in accurately finding the edges of the face.

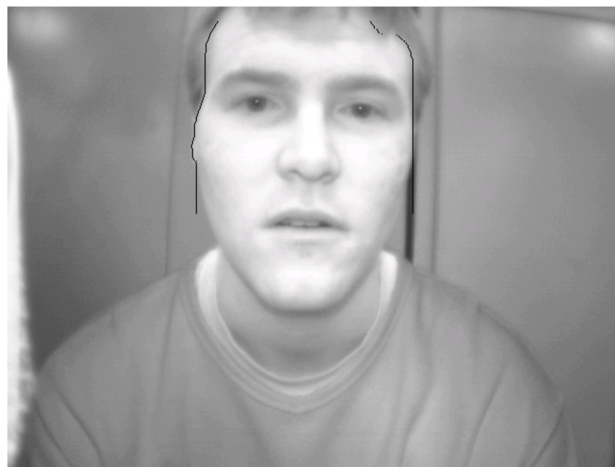


Figure 7.6: Face edges found after second trial.

Finding Intensity Changes on the Face

The next step in locating the eyes is finding the intensity changes on the face. This is done using the original image, *not* the binary image. The first step is to calculate the average intensity for each y – coordinate. This is called the horizontal average, since the averages are taken among the horizontal values. The valleys (dips) in the plot of the horizontal values indicate intensity changes. When the horizontal values were initially plotted, it was found that there were many small valleys, which do not represent intensity changes, but result from small differences in the averages. To correct this, a smoothing algorithm was implemented. The smoothing algorithm eliminated and small changes, resulting in a more smooth, clean graph.

After obtaining the horizontal average data, the next step is to find the most significant valleys, which will indicate the eye area. Assuming that the person has a uniform forehead (i.e.; little hair covering the forehead), this is based on the notion that from the top of the face, moving down, the first intensity change is the eyebrow, and the next change is the upper edge of the eye, as shown below.

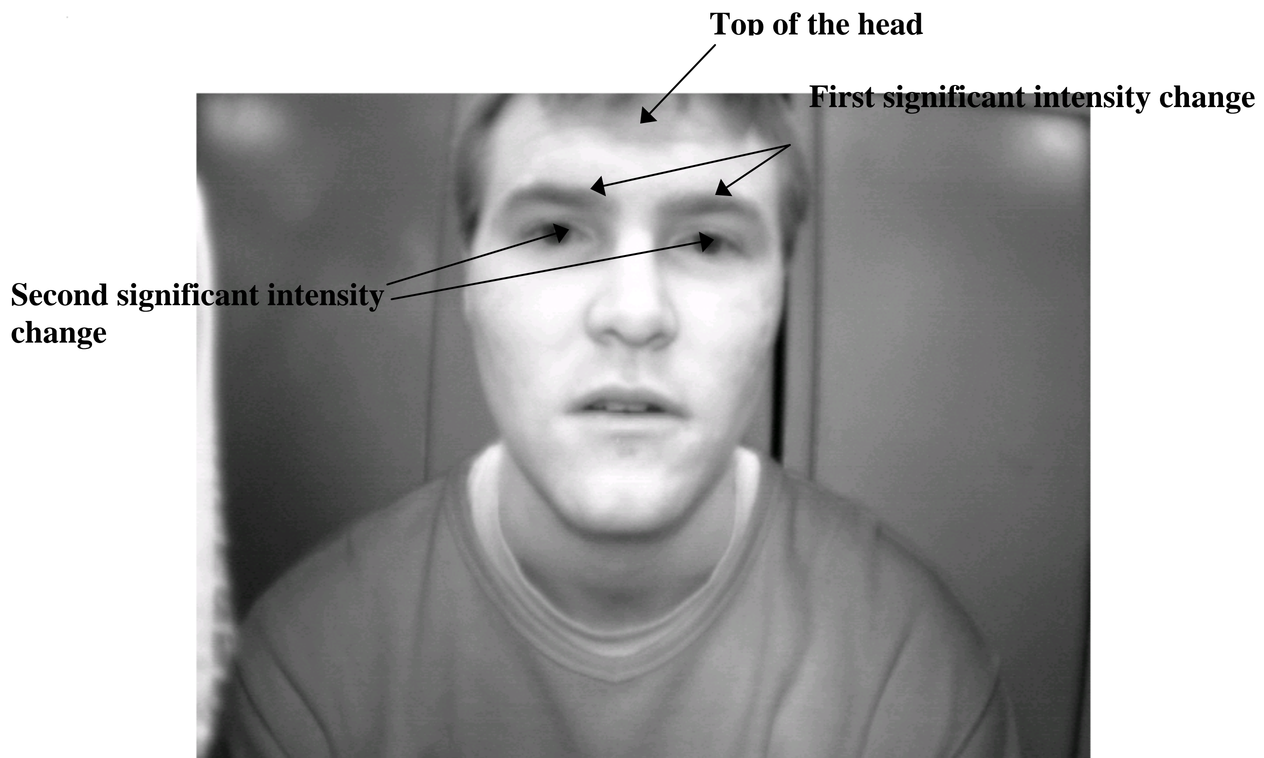
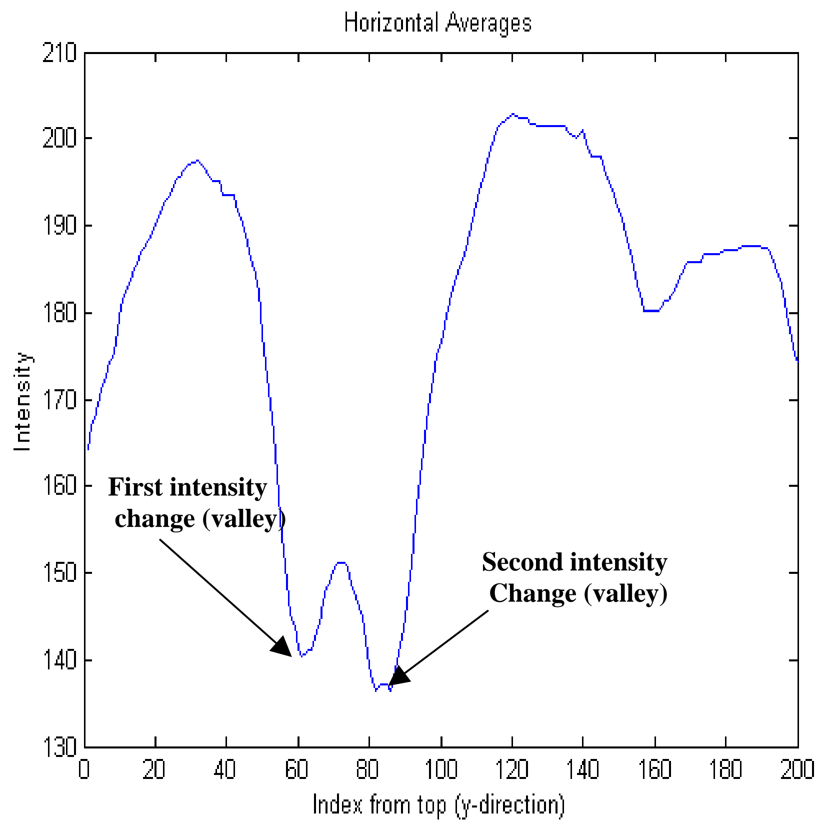


Figure 7.7: Labels of top of head, and first two intensity changes.

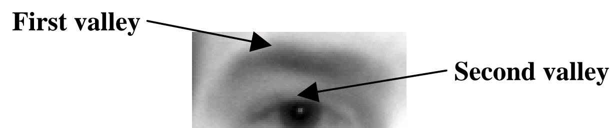
The valleys are found by finding the change in slope from negative to positive. And peaks are found by a change in slope from positive to negative. The size of the valley is determined by finding the distance between the peak and the valley. Once all the valleys are found, they are sorted by their size.

Detection of Vertical Eye Position

The first largest valley with the lowest y – coordinate is the eyebrow, and the second largest valley with the next lowest y-coordinate is the eye. This is shown in Figures 7.8a and 7.8b.



a) Graph of horizontal averages of the left side of the face.



b) Position of the left eye found from finding the valleys in a).

Figure 7.8: Result of using horizontal averages to find vertical position of the eye.

This process is done for the left and right side of the face separately, and then the found eye areas of the left and right side are compared to check whether the eyes are found correctly. Calculating the left side means taking the averages from the left edge to the centre of the face, and similarly for the right side of the face. The reason for doing the two sides separately is because when the driver's head is tilted the horizontal averages are not accurate. For example if the head is tilted to the right, the horizontal average of the eyebrow area will be of the left eyebrow, and possibly the right hand side of the forehead.

7.2.2 Drowsiness Detection Function

Determining the State of the Eyes

The state of the eyes (whether it is open or closed) is determined by distance between the first two intensity changes found in the above step. When the eyes are closed, the distance between the y – coordinates of the intensity changes is larger if compared to when the eyes are open. This is shown in Figure 7.9.

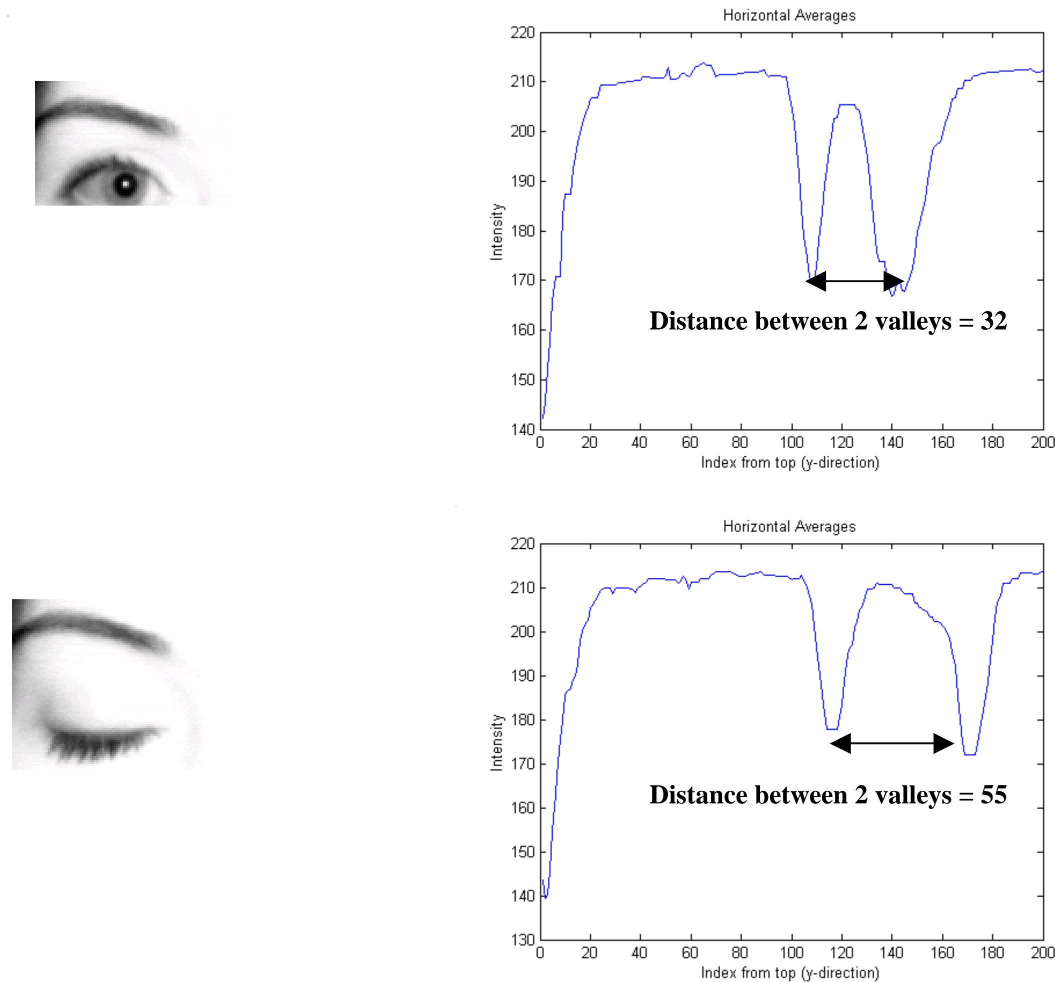


Figure 7.9: Comparison of open and closed eye.

The limitation to this is if the driver moves their face closer to or further from the camera. If this occurs, the distances will vary, since the number of pixels the face takes up varies, as seen below. Because of this limitation, the system developed assumes that the driver's face stays almost the same distance from the camera at all times.

Judging Drowsiness

When there are 5 consecutive frames find the eye closed, then the alarm is activated, and a driver is alerted to wake up. Consecutive number of closed frames is needed to avoid including instances of eye closure due to blinking. Criteria for judging the alertness level on the basis of eye closure count is based on the results found in a previous study [9].

Chapter 8

Algorithm Implementation

8.1 Real-time System

The real-time system includes a few more functions when monitoring the driver, in order to make the system more robust. There is an initialization stage, in which the for the first 4 frames, the driver's eyes are assumed to be open, and the distance between the y – coordinates of where the intensity changes occur, is set as a reference. After the initialization stage, the distances calculated are compared with the one found in the initialization stage. If the lower distance is found (difference between 5-80 pixels), then the eye is determined as being closed.

Another addition to the real-time system is a comparison of the left and right eyes found. The left and right eyes are found separately, and their positions are compared. Assuming that the driver's head is not at an angle (tilted), the y – coordinates of the left and right eye should be approximately the same. If they are not, the system determines that the eyes have not been found, and outputs a message indicating that the system is not monitoring the eyes. The system then continues to try to find the eyes. This addition is also useful in cases where the driver is out of the camera's sight. In this case, the system should indicate that no drowsiness monitoring is taking place. In the instance where there are 5 or more consecutive frames where the eye is not found, an alarm goes off. This takes account for the case when the driver's head has completely dropped down, and hence an alarm is needed to alert the driver.

8.2 Challenges

8.2.1 Obtaining the image

The first, and probably most significant challenge faced was transferring the images obtained from the camera to the computer. Two issues were involved with this challenge: 1) Capturing and transferring in real time; 2) Having access to where the image was being stored. Initially, the 'Dirt Cheap Frame Grabber' was constructed and tested. It was later found that this hardware (which uses the parallel port) would not be fast enough, and it would be difficult to write software to process the images in conjunction with the hardware.

8.2.2 Constructing an effective light source

Illuminating the face is an important aspect of the system. Initially, a light source consisting of 8 IR LEDs and a 9V battery was constructed and mounted onto the camera. It was soon realized that the source was not strong enough. After review other literature, the conclusion was that in order to build a strong enough light source, approximately 50 LEDs. To reduce the cost, a desk lamp and IR filter were used (as described previously).

8.2.3 Determining the correct binarization threshold

Because of varying facial complexions and ambient light, it was very hard to determine the correct threshold for binarization. The initial thought was to choose a value that would result in the least amount of black blobs on the face. After observing several binary images of different people, it was concluded that a single threshold that would result in similar results for all people is impossible. Histogram equalization was attempted, but resulted in no progress. Finally it was decided that blobs on the face were acceptable, and an algorithm to remove such blobs was developed.

8.2.4 Not being able to use edge detection algorithms

After reading many technical papers and image processing textbooks, it was though that locating the eyes would be a trivial task if edge detection were used. This notion turned out to be incorrect. A Sobel edge detection program was written and tested in C. Figure 8.1 shows the result. As can see, there are too many edges to find the outline of the face. Figure 8.1b shows the result of the eye area alone. It was also thought that in order to determine whether the eye was open or closed, an image processing algorithm to find circles could be used. That is, the absence of a circle would mean that the eye is closed. This also failed since many times the circle representing the iris was broken up. In addition, in order to use a circle finding algorithm, the radius of the circle must be known, and this varies depending on the distance of the driver from the camera.

a) Whole image.

b) Eye area.

Figure 8.1: Results of using Sobel edge detection.

8.2.5 Case when the driver's head is tilted

As mentioned previously, if the driver's head is tilted, calculating the horizontal averages from the left side of the head to the right side is not accurate. It is not realistic to assume that the driver's head will be perfectly straight at all times. To compensate for this, the left and right horizontal averages are found separately.

8.2.6 Finding the top of the head correctly

Successfully finding the top of the head was a big challenge. Depending on the binarization of the face, the top of the head could not be found correctly. The problem was with the black pixels on the face of the binary image. In some instances the nose, eye or lips were found to be the top of the head due to their black blobs in the binary image. Driver's with facial hair

were also often a source of this error. After implementing the noise removal algorithm, this problem was eliminated.

8.2.7 Finding bounds of the functions

By observing the system algorithm and source code, many bounds are defined. For example, when finding the edge of the face, 200 points in the y-direction is used. This represents the approximate length of the face from the top of the head (for a given distance from the camera). Another example is counting 25 points to the left and right of the centre point, when finding the edge of the face. This represents the approximate width of the face. Both of these values, along with other bounds were a challenge to determine. After examining many images of different faces, and testing different values, these bounds were determined.

Chapter 9

Results and Future Work

9.1 Simulation Results

The system was tested on 15 people, and was successful with 12 people, resulting in 80% accuracy. Figure 9.1 – 9.6 below shows an example of the step-by-step result of finding the eyes.

Figure 9.1: Demonstration of first step in the process for finding the eyes – original image.

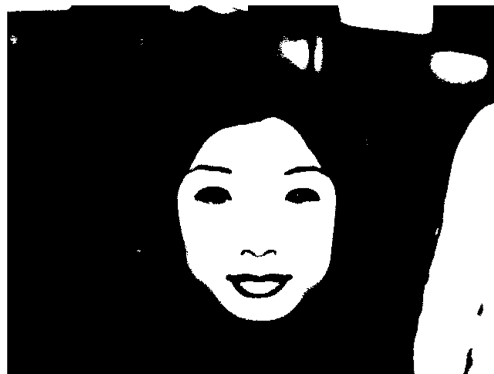


Figure 9.2: Demonstration of second step – binary image.

Figure 9.3: Demonstration of third step – initial edge detection image.

Notice that initially the edges are not found correctly.

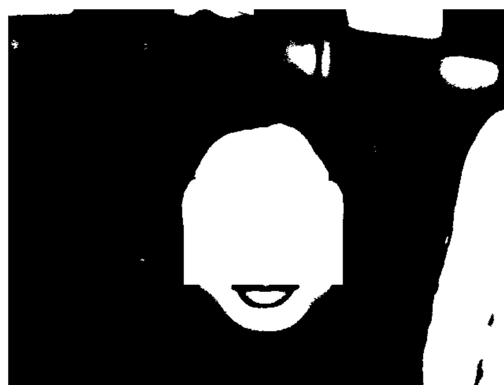


Figure 9.4: Demonstration of the fourth step - removing blobs in binary image.

Figure 9.5: Demonstration of step 5 - second edge detection.
Notice that the second time the edges are found accurately.

Figure 9.6: Result of final step - finding the left eye using intensity information.

9.2 Limitations

With 80% accuracy, it is obvious that there are limitations to the system. The most significant limitation is that it will not work with people who have very dark skin. This is apparent, since the core of the algorithm behind the system is based on binarization. For dark skinned people, binarization doesn't work.

Another limitation is that there cannot be any reflective objects behind the driver. The more uniform the background is, the more robust the system becomes. For testing purposing, a black sheet was put up behind the subject to eliminate this problem.

For testing, rapid head movement was not allowed. This may be acceptable, since it can be equivalent to simulating a tired driver. For small head movements, the system rarely loses track of the eyes. When the head is turned too much sideways there were some false alarms.

The system has problems when the person is wearing eyeglasses. Localizing the eyes is not a problem, but determining whether the eyes are opened or closed is.

9.3 Future Work

Currently there is not adjustment in zoom or direction of the camera during operation. Future work may be to automatically zoom in on the eyes once they are localized. This would avoid the trade-off between having a wide field of view in order to locate the eyes, and a narrow view in order to detect fatigue.

This system only looks at the number of consecutive frames where the eyes are closed. At that point it may be too late to issue the warning. By studying eye movement patterns, it is possible to find a method to generate the warning sooner.

Using 3D images is another possibility in finding the eyes. The eyes are the deepest part of a 3D image, and this maybe a more robust way of localizing the eyes.

Adaptive binarization is an addition that can help make the system more robust. This may also eliminate the need for the noise removal function, cutting down the computations needed to find the eyes. This will also allow adaptability to changes in ambient light.

The system does not work for dark skinned individuals. This can be corrected by having an adaptive light source. The adaptive light source would measure the amount of light being reflected back. If little light is being reflected, the intensity of the light is increased. Darker skinned individual need much more light, so that when the binary image is constructed, the face is white, and the background is black.

Chapter 10

Conclusion

A non-invasive system to localize the eyes and monitor fatigue was developed. Information about the head and eyes position is obtained though various self-developed image processing algorithms. During the monitoring, the system is able to decide if the eyes are opened or closed. When the eyes have been closed for too long, a warning signal is issued. In addition, during monitoring, the system is able to automatically detect any eye localizing error that might have occurred. In case of this type of error, the system is able to recover and properly localize the eyes. The following conclusions were made:

- Image processing achieves highly accurate and reliable detection of drowsiness.
- Image processing offers a non-invasive approach to detecting drowsiness without the annoyance and interference.
- A drowsiness detection system developed around the principle of image processing judges the driver’s alertness level on the basis of continuous eye closures.

All the system requirements described in Chapter 2 were met.

Appendix A

Program Listing

```
/*
 * ddds.c -
 *
 * Drowsy Driver Detector System Source Code
 * By: Neeta Parmar
 * Ryerson University
 * Department of Electrical and Computer Engineering
 * Copyright 2002
 */

/*
 * Aim:
 * To localize the eyes from an image of a person's face, and then
 * monitor whether the eye is opened or closed.  Five consecutive
 * frames of eye closure means that the driver is sleeping, and hence
 * an alarm is generated to alert the drowsy driver.
 */

/* The following next paragraphs describes each function, and how the
 * are used in the Drowsy Driver Detection System.
 * See thesis report for entire overview of the system.
 */

/* Function: void Picolo_initialize()
 *
 * Purpose: To initialize the Picolo frame grabber board.
 *
 * It uses some predefined structures and functions to initialize the
 * Picolo library and board, and allocate memory for the acquire image.
 * The video source consists of a BNC composite video input, and the
 * video signal type is NTSC.  The function also defines the image
 * format, in this case the image format used is greyscale
 * (PICOLO_COLOR_Y8). The width of the image is 640 pixels and the
 * height is 240 pixels.
 */
```

```
/* Function: void Acquire()
 *
 * Purpose: To capture the image and store the image values into an
 * array, Pic.
 *
 * Initially, the Picolo library stores the image into its predefined
 * buffer, and so this buffer is copied into the Pic array. The Pic
 * array will be used in most of the other functions, and is defined as
 * a global variable.
 */


/* Function: void Binarization()
 *
 * Purpose: To convert the image into a binary picture.
 *
 * Based on a predefined threshold, the image is binarized into two
 * values, black and white.  If a pixel in the Pic array is less than
 * or equal to the threshold it is set to black, if greater, set to
 * white.  The binary image is stored in the BWPic array.
 */


/* Function: int Face_detect()
 *
 * Purpose: To find an accurate centre of the face (in the x-
 * direction).
 *
 * Starting at the point (100,240) of the binary image, the edges of
 * the face are found along the x-direction.  When the left and right
 * edges are found, the centre of the two is assigned to x1.  This
 * value (x1) is returned, and is later used as the new centre of the
 * face.
 */


/* Function: int Top_detect(int x, int y)
 *
 * Purpose: To find the top of the face.
 *
 * This function passes two integers.  The first, x, is the x-
 * coordinate corresponding to the centre, obtained from either the
 * output of Face_detect or Detect_centre depending on at which point
 * in the algorithm it is being called. Similarly, y is the y-
 * coordinate corresponding to the centre of the face.
 * The top is detected by decrementing the y-coordinate, keeping track
 * of the number of black pixels found. The function returns the top
 * value (y-coordinate).
 */
```

```
/* Function: int Detect_centre (int x, int y)
 *
 * Purpose: To find the centre of the face.
 *
 * This function passes two integers.  The first integer is the centre
 * x-coordinate, the second is the y-coordinate representing the top of
 * the head.  Starting at the pixel value of the two passed
 * coordinates, the left and right sides of the face are found by
 * looking for a change in pixels value from a white to black.
 * The centre is the mid-point of the left and right side.  This value
 * is returned back (x-coordinate).
 */


/* Function: int Detect_edge(int x, int y)
 *
 * Purpose: To find the left and right edges of the face.
 *
 * This functions passes two integers.  They are the x and y
 * coordinates for the top of the head.  The y coordinate is
 * decremented and the left and right sides are found as described in
 * Detect_center. The y coordinate is decremented 200 times which
 * corresponds to an approximate length of the face.  The left and right
 * pixel values of the sides are stored in two separate arrays. A new
 * centre value of the face is returned.
 */


/* Function: void Remove_noise()
 *
 * Purpose: To remove the black blobs in the binary picture.
 *
 * Starting at the pixel value of the top of the head,(and centre in
 * the x-direction),the left side and then right side of the face are
 * set to white. This is only done up to the left and right edges found
 * in Detect_edge.
 */


/* Function: void Horz_avg (int *start, int *end, int side)
 *
 * Purpose: To calculate the horizontal averages.
 *
 * The function passes three parameters.  The first two are the arrays
 * containing the left and right pixel values of the sides found in
 * Detect_edge.  The third parameter is an integer representing which side
 * of the face the calculations are being done on (left = 1; right = 2).
 * A smoothing algorithm is implemented to remove any small changes in the
 * horizontal values so they are not latter confused as being valleys.
 */
```

```
/* Function: int Find_valley(int side)
 *
 * Purpose: To identify the valleys among the horizontal averages.
 *
 * This function passes an integer to indicate the side of the face (see
 * above explanation).  The valleys are found based on slope changes, and
 * the size of a valley is defined by taking the distance between a valley
 * and peak (in the y-direction.  This function also sorts the valley
 * according to their size.  The number of valleys found, ind, is
 * returned.


/* Function: int Find_index(int ind, int side)
 *
 * Purpose: To find the indexes (y-coordinates) of the vertical position
 * of the eye.
 *
 * This function using the valleys found in the previous function to find
 * the vertical positions of the eyes.  The parameters passed are the
 * number of valleys and the integer indicating which side is being
 * processed.  Once the two vertical positions of the eyes are found
 * (usually the eyebrow and upper eye), the starting and ending areas of
 * the eye are defined.  This is done using only the second index.  The
 * difference of the two indexes is a measurement of the eye, and is
 * returned.  This value is used to determine the state of the eye.
 */


/* Function: void Printeye(int side, int f)
 *
 * Purpose: To store the eye data into a text file
 *
 * This function stores the eye data into a text file.  The first value
 * Passed indicates which eye (left or right) is stored, and the second
 * value indicates the file number.  The files can then be loaded into
 * Matlab to view the eye image.
 */
```

```
/* ------------------------- START OF PROGRAM ------------------------ */


#include <owl/opensave.h>
#include <picolo32.h>
#include <E_type.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>

#define Black 0
#define White 255
#define Negative 0
#define Positive 1
#define HEIGHT 480
#define WIDTH 640
#define ITERATIONS 200

PICOLOHANDLE hPicolo, hBuffer;
UINT32 ImageWidth, ImageHeight, ImageSize;
BYTE *pun8ImageBuffer,*pun8AlignedBuffer;
PICOLOSTATUS PicoloStatus;

int fb = 0;
int f = 0;
int x,y;
int pic[640*480];
int BWPic[640][480];
int Pic[640][480];
int Left_x[200];
int Right_x[200];
int Vertical_y[200];
int ind;
float Horz_average_l[200];
float Horz_average_r[200];
float Valley [30];
float Valley_sort[30];
int Index[30];
int Index1[30];
int Index2[30];
int eye_x_start_l,eye_x_end_l,eye_y_start_l,eye_y_end_l;
int eye_x_start_r,eye_x_end_r,eye_y_start_r,eye_y_end_r;
int Top;
int index1_left,index1_right,index2_left,index2_right;
int distance_left, distance_right;
int Top_x, Top_y;
int open_dist,distance_new;
int distance_old;

FILE *stream, *fd, *fd1,*fd2;
```

```
/* Function prototypes */

void Picolo_initialize();
void Acquire();
void Binarization();
int Face_detect();
int Top_detect(int x, int y);
void Detect_edge(int x, int y);
void Remove_noise(int x1, int Top);
void Horz_avg(int *start , int *end, int side);
int Find_valley(int side);
int Find_index(int ind, int side);
int Detect_center( int x, int y);
void Printeye(int side, int f);

main(){

distance_old=0;
open_dist =0;
f=0;
int i,j;

/* Initialize the Picolo hardware */

Picolo_initialize();
printf("START:");
getchar();


/* For the first 4 frames, get open eye information: this will be used
 * as a reference to determine whether the eye is open or closed.  The
 * variable distance_old will be the reference distance.
 */

for (int z=1; z<5;z++){
      Acquire();
      Binarization();
      x = Face_detect();
      Top_y = Top_detect(x, 240);
      y = Top_y +10;
      Top_x = Detect_center(x,y);
      Top_y = Top_detect(Top_x, y);
      Detect_edge(Top_x, Top_y );
      Remove_noise(Top_x,Top_y);
      Detect_edge(Top_x, Top_y );

      /* Left side calulations */

      Horz_avg(Left_x, Right_x,1);
      ind = Find_valley(1);
      distance_left = Find_index(ind,1);
```

```
	/* Right side calculations */

	Horz_avg(Left_x, Right_x,2);
	ind =Find_valley(2);
	distance_right = Find_index(ind,2);


	/* Check to see if the coordinates found for the left and right eye
	 * are similar.  If so, store left eye data so that it can be view
	 * in Matlab. If left and right eye positions do not match, output
	 * error message and store junk data in second folder.
	 */

	if (abs(index2_right-index2_left <25)){
		Printeye(1,f);
		if (f<80)f++;
	}
	else {
		printf("ERROR -- Eye not found\n");
		distance_left = 0;
	}

	distance_new= distance_left;


	/* If distance_new = 0, this means that the eye was not found
	 * correctly. In this case, set z = 1: do not leave the loop until
	 * the eye is found correctly.  This ensures that when determining
	 * whether the eye is open closed, a correct open eye information is
	 * being used.
	 */

	if (distance_new == 0){
		z=1;
		open_dist = 0;
	}

	else {
		open_dist = open_dist + distance_new;
		distance_old =  open_dist/z;
	}
} /* End of obtaining initial open eye information */
```

```
/* After obtaining the initial open eye information, continue capturing
 * images, monitoring the state of the eye.
 */

for(;;){

      Acquire();
      Binarization();
      x = Face_detect();
      Top_y = Top_detect(x, 240);
      y = Top_y+10;
      Top_x = Detect_center(x,y);
      Top_y = Top_detect(Top_x, y);
      Detect_edge(Top_x, Top_y );
      Remove_noise(Top_x,Top_y);
      Detect_edge(Top_x, Top_y );


      /* Left side calulations */

      Horz_avg(Left_x, Right_x,1);
      ind = Find_valley(1);
      distance_left = Find_index(ind,1);

      /* Right side calulations */

      Horz_avg(Left_x, Right_x,2); //Right side calculations
      ind =Find_valley(2);
      distance_right = Find_index(ind,2);


      /* Check to see if the coordinates found for the left and right eye
       * are similar.  If so, store left eye data so that it can be view
       * in Matlab.  If left and right eye positions do not match, output
       * error message and store junk data in second folder.
       */


      if (abs(index2_right-index2_left <25)){
            Printeye(1,f);
            f++;
      }
      else {
            printf("ERROR -- Eye not found\n");
            distance_left = 0;
      }

      distance_new= distance_left;
```

```
/* Determine the state of the eye, by comparing it to the distance found
 * in the first four frames. If distance_new = 0, this means that the eye
 * was not found correctly, therefore this step is not done.  If the
 * difference between the new distance and the old distance is between 5
 * and 80, then the eye is closed, and alarm will be issued.
 */

      if (distance_new!= 0){
            if  (distance_new  >  distance_old){
                  if (((distance_new - distance_old) > 5) &&
                        ((distance_new - distance_old) < 80)){
                              putchar('\a');
                              printf("EYES CLOSED\n\n");
                  }
            }
      }
} /* End of for loop */
} /* End of main */


/* Picolo intialization function: refer to comments at the beginning for
this function */

void Picolo_initialize(){

// --------------------------------------------------------------------------
// Picolo: Initialize PICOLO library
// --------------------------------------------------------------------------

hPicolo = PicoloStart(0);           // use board #0
if (hPicolo < 0){
      printf("Cannot start PICOLO");
      goto finish;
}



// --------------------------------------------------------------------------
// Picolo: Define video source
// --------------------------------------------------------------------------
PicoloStatus = PicoloSelectVideoInput(hPicolo, PICOLO_INPUT_COMPOSITE_BNC,
PICOLO_IFORM_STD_NTSC);
if (PicoloStatus != PICOLO_OK){
      printf("Cannot select the Video Input");
      goto finish;
}
// --------------------------------------------------------------------------
// Picolo: Select the Image format
// --------------------------------------------------------------------------
PicoloStatus = PicoloSelectImageFormat(hPicolo, PICOLO_COLOR_Y8);
if (PicoloStatus != PICOLO_OK){
      printf("Cannot select image format");
      goto finish;
}
```

```
// ---------------------------------------------------------------------------
// Picolo: Get Image Width and Height
// ---------------------------------------------------------------------------
PicoloStatus = PicoloGetImageSize(hPicolo, &ImageWidth, &ImageHeight);
if(PicoloStatus != PICOLO_OK){
      printf("Call to PicoloGetImageSize() unsuccessfull");
      goto finish;
}

// ---------------------------------------------------------------------------
// Picolo: Allocate an image buffer for a full frame image
// ---------------------------------------------------------------------------
UINT32 un32ImageBufferSize;
PicoloGetImageBufferSize(hPicolo, & un32ImageBufferSize);

pun8ImageBuffer = (PUINT8)VirtualAlloc(NULL, un32ImageBufferSize,
MEM_COMMIT, PAGE_READWRITE);

if (pun8ImageBuffer == NULL){
      printf("Cannot allocate image buffer");
      goto finish;
}

// ---------------------------------------------------------------------------
// Picolo: Register the transfer buffer
// ---------------------------------------------------------------------------
hBuffer = PicoloSetImageBuffer(hPicolo, pun8ImageBuffer,
un32ImageBufferSize, 0,(void **)&pun8AlignedBuffer);

if (hBuffer < 0){
      printf("Cannot set image buffer");
      goto finish;
}

finish:
}   /* End of Picolo_Initialization */



/* Image Acquire function: refer to comments at the beginning for this
function */

void Acquire(){

int inbuff[640*480];
int size, x, y, i ;

size = WIDTH*HEIGHT;

PicoloStatus = PicoloAcquire(hPicolo, PICOLO_ACQUIRE, 1);

if (PicoloStatus < 0){
      printf("Cannot grab");
      goto finish;
}
```

```
for (i=0; i< size; i++)
      inbuff[i]=255;

for (i =0; i< size; i++)
      inbuff[i]= pun8ImageBuffer[i];

for (y=0; y<HEIGHT; y++){
      for (x=0; x< WIDTH; x++){
      Pic[x][y]=inbuff[WIDTH*y+x];
   }
}

//Free buffer
PicoloReleaseImageBuffer(hPicolo,hBuffer);

// ----------------------------------------------------------------------
// Picolo: Stop PICOLO driver
// ----------------------------------------------------------------------

PicoloStop(hPicolo);
finish:
}     /* End of Acquire */



/* Image binarization function: refer to comments at the beginning for
this function */

void Binarization(){

int threshold = 150;
int i,j;

for(i=0;i<HEIGHT;i++){
      for(j=0;j<WIDTH;j++){
            if (Pic[j][i] <= threshold){
                  BWPic[j][i]= Black;
            }
            else
            BWPic[j][i] = White;
      }
}

}     /* End of Binarization */
```

```
/* This function finds the middle point of the face (in the x -
 * direciton). Refer to comments at the beginning for this function
 */

int Face_detect(){

int i, j, x, y, x1;
y= 240;
x=100;

/* White detect looks for an initial white pixel, and counts upto 25 more
 * consectutive white pixels to determine whether a white edge is found.
 * If in between the 25 pixels there is a black pixel, start counting from
 * the beginning.
 */

White_detect: while (BWPic[x][y]==Black){x++;}

              /* Assuming y centre is in face */
              for (j=0; j<25; j++){
                  if (BWPic[x+j][y] == Black){
                        x=x+j;
                        goto  White_detect;
                  }
              }

x1 = x;


/* Black detect looks for an initial black pixel, and count upto 25 more
 * consectutive black pixels to determine whether a black edge is found.
 * If in between the 25 pixels there is a white pixel, start counting from
 * the beginning.
 */

Black_detect: while (BWPic[x][y]==White){x++;}
                    for (j=0; j<25; j++){
                        if (BWPic[x+j][y]==White){
                              x=x+j;
                              goto  Black_detect;
                        }
                    }

x1=int(x1+(x-x1)/2);  /* x is midle point of two boundries */
return (x1);
}     /* End of Face_detect */
```

```
/* This function finds the left and right edges of the face.
 * Refer to comments at the beginning for this function.
 */

void  Detect_edge(int x, int y){

int i, j;

/* Find the left and right edges of the face for 200 y values. */

for (i=0; i<ITERATIONS; i++){
      y++;

      /* Detect the right side of the face by looking for a white pixel
       * followed by 5 consectutive black pixels.
       */

      Right_detect: while (BWPic[x][y] == White) {x++;}
                    for (j=0; j<5; j++){
                         if (BWPic[x+j][y] != Black){
                                x=x+j;
                                goto  Right_detect;
                         }
                    }
   Right_x[i]=x;  /* Store the right edge pixel value. */
   x--;


      /* Detect the left side of the face by looking for a black pixel
       * followed by 5 consectutive white pixels.
       */

      Left_detect: while (BWPic[x][y] == White) {x--;}
                   for (j=0; j< 5; j++){
                         if (BWPic[x-j][y] != Black){
                                x=x-j;
                                goto  Left_detect;
                         }
                   }
   Left_x[i]=x;          /* Store the left edge pixel value */
   Vertical_y[i]=y;
   x=((Right_x[i]-Left_x[i])/2)+ Left_x[i];  /* Center point value */
   x1=x;    /* New top value */
}

}     /* End of Detect_edge */
```

```
/* This function removes the black blobs on the binary image of the face.
 * Refer to comments at the beginning for this function.
 */

void Remove_noise(int x1, int Top){

int i, k ,y;

for (k=0; k<100; k++){
	if (Left_x[k+1] < Left_x[k]){
		for (i=Left_x[k]; i>Left_x[k+1]; i--){
			for (y=Top_y+k; y<Top_y+ITERATIONS;y++){
				BWPic[i][y]=White;
			}
		}
	}
	else{
		for (i=Left_x[k]; i<Left_x[k+1]; i++){
			for (y=Top_y+k; y<Top_y+ITERATIONS;y++){
				BWPic[i][y]=White;
			}
		}
	}
}

for(i=x1; i>Left_x[0]; i--){
	for (y=Top_y; y<Top_y+ITERATIONS;y++){
		BWPic[i][y]=White;
	}
}

for (k=0; k<100; k++){
	if (Right_x[k+1] < Right_x[k]){
		for (i=Right_x[k]; i>Right_x[k+1]; i--){
			for (y=Top_y+k; y<Top_y+ITERATIONS;y++){
				BWPic[i][y]=White;
			}
		}
	}
	else{
		for (i= Right_x[k]; i<Right_x[k+1]; i++){
			for (y=Top_y+k; y<Top_y+ITERATIONS;y++){
				BWPic[i][y]=White;
			}
		}
	}
}


for(i=x1; i<Right_x[0]; i++){
	for (y=Top_y; y<Top_y+ITERATIONS;y++){
		BWPic[i][y]=White;
	}
}
}	/* End of Remove_noise */
```

```
/* This function defines the centre of the face.
 * Refer to comments at the beginning for this function.
 */

int Detect_center( int x, int y){

int j, rx, lx;

Right_detect: while (BWPic[x][y] == White) {x++;}
              for (j=0; j<25; j++){
                   if (BWPic[x+j][y] != Black){
                         x=x+j;
                         goto  Right_detect;
                   }
              }
rx =x; /* Right side of face */
x--;

Left_detect: while (BWPic[x][y] == White) {x--;}
             for (j=0; j< 25; j++){
                   if (BWPic[x-j][y] != Black){
                         x=x-j;
                         goto  Left_detect;
                   }
             }
lx=x; /* Left side of face */
x=((rx-lx)/2)+ lx;  /* Center point value */
return (x);
}     /* End of Detect_center */



/* This function finds the top of the head.
 * Refer to comments at the beginning for this function.
 */


int Top_detect(int x, int y){

int i=0;
int k=0;

Detect_top: while(BWPic[x][y] != Black) {
                  y--;
                  if (y<=0) break;  /* If y reaches top of picture */
            }

            while((BWPic[x][y-i] == Black)&& (y-i)>0 && i<120){I++;}

            if ((i<120) && (y-i)>0){
                  y = y - i;
                  goto Detect_top;
            }
return (y);
}
```

```
/* This function calculates the average intesnsity for each horizantal
 * line. Refer to comments at beginning for this function.
 */

void Horz_avg(int *start , int *end, int side){

int Sum =0;
float Horz_average [ITERATIONS];
int i,j;

/* Left side calculations */
if (side==1){
      for (j=0; j<ITERATIONS; j++){
            for (i=start[j]; i<(start[j]+(end[j]- start[j])/2); i++) {
                  Sum = Sum + Pic[i][Vertical_y[j]];
            }
      Horz_average_l[j]= Sum/((end[j] - start[j])/2); /* Left side only */
      Sum = 0;
   }
}
/* Right side calculations */
if (side==2){
      for (j=0; j<ITERATIONS; j++){
            for (i=(start[j]+(end[j]- start[j])/2); i< end[j]; i++) {
                  Sum = Sum + Pic[i][Vertical_y[j]];
            }
      Horz_average_r[j]= Sum/((end[j] - start[j])/2); /* Right side only*/
      Sum = 0;
   }
}
/* Smooth the horizontal line average curve */

/* Left side */

for (j=0;j<ITERATIONS-3;j++){
      if ((Horz_average_l[j+1] - Horz_average_l[j] >= 0) &&
         (Horz_average_l[j+2] - Horz_average_l[j+1] < 0) &&
         (Horz_average_l[j+3] - Horz_average_l[j+2] >= 0)){
            Horz_average_l[j+1] = Horz_average_l[j];
            Horz_average_l[j+2] = Horz_average_l[j];
      }

      if ((Horz_average_l[j+1] - Horz_average_l[j] <= 0) &&
         (Horz_average_l[j+2] - Horz_average_l[j+1] > 0) &&
         (Horz_average_l[j+3] - Horz_average_l[j+2] <= 0)){
            Horz_average_l[j+1] = Horz_average_l[j];
            Horz_average_l[j+2] = Horz_average_l[j];
      }
}
```

```
/* Right side */

for (j=0;j<ITERATIONS-3;j++){
      if ((Horz_average_r[j+1] - Horz_average_r[j] >= 0) &&
         (Horz_average_r[j+2] - Horz_average_r[j+1] < 0) &&
         (Horz_average_r[j+3] - Horz_average_r[j+2] >= 0)){
            Horz_average_r[j+1] = Horz_average_r[j];
            Horz_average_r[j+2] = Horz_average_r[j];
      }

      if ((Horz_average_r[j+1] - Horz_average_r[j] <= 0) &&
         (Horz_average_r[j+2] - Horz_average_r[j+1] > 0) &&
         (Horz_average_r[j+3] - Horz_average_r[j+2] <= 0)){
            Horz_average_r[j+1] = Horz_average_r[j];
            Horz_average_r[j+2] = Horz_average_r[j];
      }
}

}     /* End of Horz_avg */


/* This function finds the valleys in the horizantal average data.
 * Refer to comments at beginning for this function.
 */

int Find_valley(int side){

int slope,ind;
float Peak[30];
float peak;
float  Horz_average [ITERATIONS];
int index_next_small;
float next_small,temp;
int i, j, k;
ind = 0;

/* Initialize the arrays. */

for (j=0;j<30;j++){
      Valley[j]=0;
   Valley_sort[j]=0;
   Peak[j]=0 ;
   Index[j]=0;
   Index1[j]=0;
   Index2[j]=0;
}

/* If doing left side calculations. */

if (side== 1){ for (i=0; i<ITERATIONS;
i++)Horz_average[i]=Horz_average_l[i];}
```

```
/* If doing right side calculations. */

if (side ==2){ for (i=0;i<ITERATIONS;
i++)Horz_average[i]=Horz_average_r[i];}

/* Find the peaks and valleys, and the distances between each peak and
valley. */

/* Set the peak to the first average value, and the slope to positive. */

peak =Horz_average[0];
slope =Positive;


for (j=0;j<ITERATIONS;j++){
      if ((Horz_average[j+1] - Horz_average[j] >0) &&
         (slope == Negative)){
            Valley[ind] = peak - Horz_average[j];
            Index[ind]=j;
            peak = Horz_average[j+1];
            Peak[ind]=peak;
            slope = Positive;
            ind = ind+1;
      }

      if (Horz_average[j+1] - Horz_average[j] > 0){
            peak = Horz_average[j+1];
            slope =Positive;
      }

      if (Horz_average[j+1] - Horz_average[j] < 0){
            slope = Negative;
      }
}

/* Copy Valley array to a second array. */

for (j=0;j<ind;j++){ Valley_sort[j] = Valley[j]; }

/* Sort the Valley array to find valleys with greatest distance (from peak
to valley). */

for (j=0;j<ind-1;j++){
      next_small = Valley_sort[j];
      index_next_small = j;

      for(k=j;k<ind;k++){
            if(Valley_sort[k] > next_small){
                  next_small = Valley_sort[k];
                  index_next_small = k;
            }
      }
      temp = Valley_sort[j];
      Valley_sort[j] = Valley_sort[index_next_small];
      Valley_sort[index_next_small] = temp;
```

```
}
return (ind);
}     /* End of Find_valley */


/* This function finds the vertical position of the eye based on the
 * valley information. Refer to comments at beginning for this function.
 */

int Find_index(int ind, int side){

int j,k,temp;
int index1 = 0;
int index2 = 0;
int count=0;
int distance;

/* Find the corresponding indexes (y-coordinate values) for each valley */

for (j=0;j<=ind;j++){
      for(k=0;k<=ind;k++){
            if (Valley_sort[k]==Valley[j]){
                  Index1[k]=Index[j];
            }
      }
}

Index2[0]=0;

/* After obtaining the y - coordinates for each valley, the position of
 * the eye is found based on two facts: The lowest index is the first y-
 * coordinate of the eye, and the next lowest is the second coordinate.
 * Keep in mind, that these indexes represent to greatest valleys, which
 * represent large intensity changes in the face.
 */

if (Index1[0]>Index1[1]){
      index1 = Index1[1] + Top_y;

      /* Switch first and second indexes, so that Index1[1] is not
       * mistakenly considered as the second y - coordinate of the eye.
       */

      temp=Index1[0];
      Index1[0]=Index1[1];
      Index1[1]=temp;
```

```
   /* Of the remaining indexes (y - coordinates), the next index should be
    * within 65 points of the first index.  Create a new array only
    * containing these values.
    */

      for (j=0;j<ind;j++){
            if ((Index1[j] - (index1-Top_y) < 70)&&
               (Index1[j] - (index1-Top_y) > 5)){
                  Index2[count] = Index1[j];
                  count++;
            }
      }
}


if (Index1[0]<Index1[1]){
      index1= Index1[0] + Top_y;
      for (j=0;j<ind;j++){
            if ((Index1[j] - (index1-Top_y) <70)&&
               (Index1[j] - (index1-Top_y) > 5)){
                  Index2[count] = Index1[j];
                  count++;
            }
      }
}

/* Of the new array created after finding the first index, the second
 * index must be the first content in the new array.
 */

index2 = Index2[0] + Top_y;

/* Print only eye area starting form second valley and to second valley
 * +15.
 */

if (side==1){
      eye_y_start_l = index2-10;
      eye_y_end_l   = index2 + 15;
      eye_x_start_l = Left_x[index2-Top_y-10];
      eye_x_end_l   = Right_x[index2-Top_y+15];

      /* Center point of the being and end */

      eye_x_end_l =  ((eye_x_end_l - eye_x_start_l)/2)+ eye_x_start_l;  }

if (side ==2){
      eye_y_start_r = index2-10;
      eye_y_end_r   = index2 + 15;
      eye_x_start_r = Left_x[index2-Top_y-10];
      eye_x_end_r   = Right_x[index2-Top_y+15];

      eye_x_start_r =  (eye_x_end_r - (eye_x_end_r - eye_x_start_r)/2);
}
```

```
distance = index2-index1;

if (side ==1){
      index1_left = index1;
      index2_left = index2;
}

if (side == 2){
      index1_right = index1;
      index2_right = index2;
}

return (distance);
} /* End of Find_index */



/* This function stores the data of the eye area in a text file so that it
 * can be viewed in Matlab. Refer to comments at beginning for this
 * function.
 */

void Printeye(int side, int f){

int i, j;
char num[2];
char ext[4] = ".txt";
char filename[22] = "d:/neeta/data/eye/";

itoa(f,num,10);
if (f<10)
      strncat(filename,num,1);
if (f>=10)
      strncat(filename,num,2);

strncat(filename,ext,4);

fd= fopen(filename, "w+");

if (side == 1){
   for(i=eye_y_start_l;i<eye_y_end_l;i++){
      for(j=eye_x_start_l;j<eye_x_end_l;j++){
              fprintf(fd, "%d\t ", Pic[j][i]);
      }
      fprintf(fd,"\n");
   }
}

if (side == 2){
   for(i=eye_y_start_r;i<eye_y_end_r;i++){
      for(j=eye_x_start_r;j<eye_x_end_r;j++){
              fprintf(fd, "%d\t ", Pic[j][i]);
      }
      fprintf(fd,"\n");
```

```
   }
}
fclose(fd);

}     /* End of Printeye */


/* -------------------------- END OF PROGRAM ------------------------- */
```

Bibliography

[1] Davies, E.R. "Machine Vision: theory, algorithms, and practicalities", *Academic Press:* San Diego, 1997.

[2] Dirt Cheap Frame Grabber (DCFG) documentation, file dcfg.tar.z available from http://cis.nmclites.edu/ftp/electronics/cookbook/video/

[3] Eriksson, M and Papanikolopoulos, N.P. "Eye-tracking for Detection of Driver Fatigue", *IEEE Intelligent Transport System Proceedings* (1997), pp 314-319.

[4] Gonzalez, Rafel C. and Woods, Richard E. "Digital Image Processing", *Prentice Hall:* Upper Saddle River, N.J., 2002.

[5] Grace R., et al. "A Drowsy Driver Detection System for Heavy Vehicles", *Digital Avionic Systems Conference, Proceedings, 17th DASC. The AIAA/IEEE/SAE, I36/1-I36/8* (1998) vol. 2.

[6] Perez, Claudio A. et al. "Face and Eye Tracking Algorithm Based on Digital Image Processing", *IEEE System, Man and Cybernetics 2001 Conference,* vol. 2 (2001), pp 1178-1188.

[7] Singh, Sarbjit and Papanikolopoulos, N.P. "Monitoring Driver Fatigue Using Facial Analysis Techniques", *IEEE Intelligent Transport System Proceedings* (1999), pp 314-318.

[8] Ueno H., Kanda, M. and Tsukino, M. "Development of Drowsiness Detection System",*IEEE Vehicle Navigation and Information Systems Conference Proceedings*, (1994), ppA1-3,15-20.

[9] Weirwille, W.W. (1994). "Overview of Research on Driver Drowsiness Definition and Driver Drowsiness Detection," *14th International Technical Conference on Enhanced Safety of Vehicles,* pp 23-26.