TBD

Names go here

First printing, July 2014

Contents

Preface

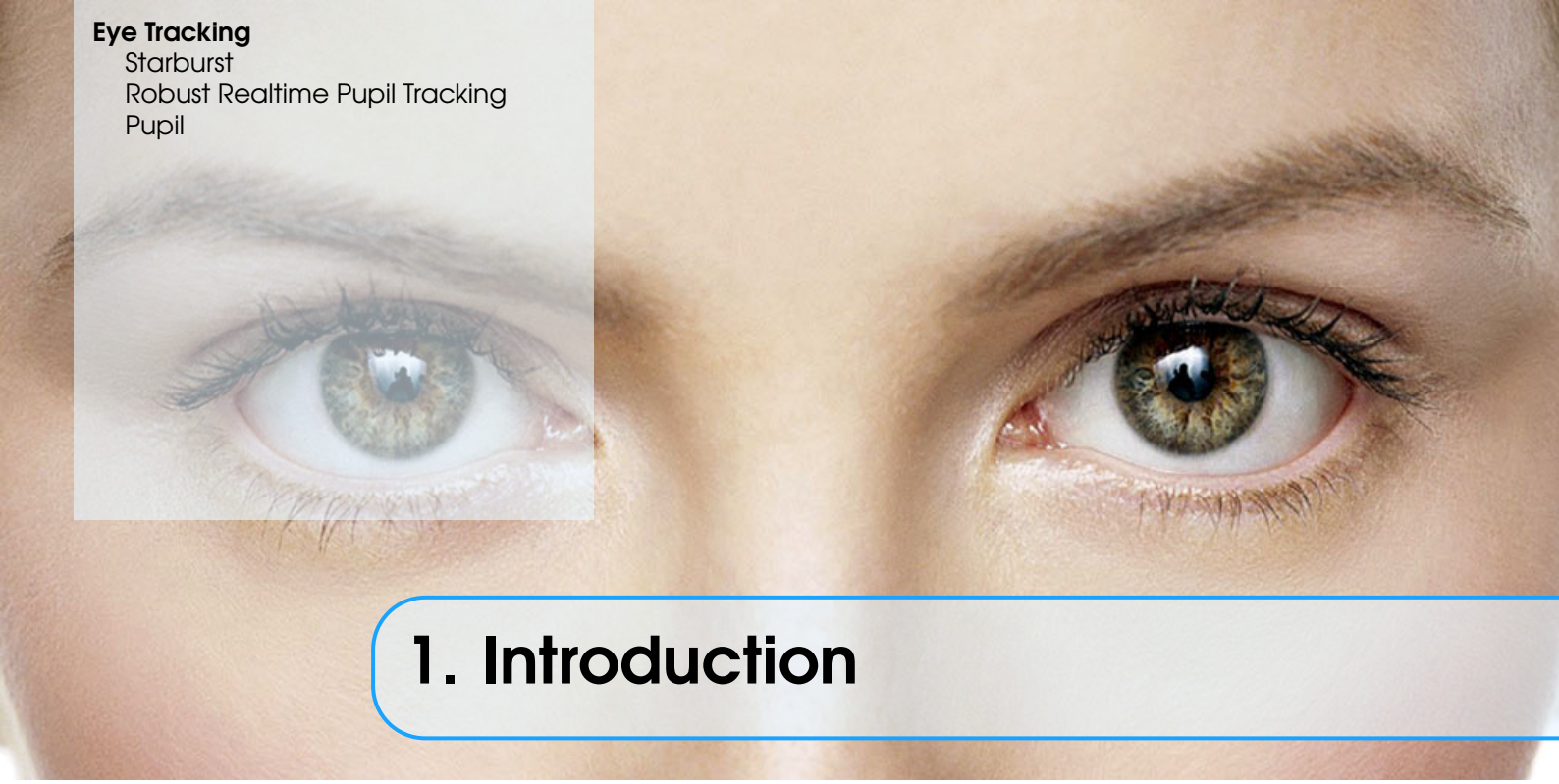
Abstract

With the ever-increasing diffusion of wearable computers in our lives, and the increasing time we spend using such devices, developing new techniques that ease interaction with computers has become insistent. One of the most interesting topics in this field is how to allow user to interact with computers without the traditional mode of interaction (mouse, keyboard and even touch-screen). We believe that eye tracking and gesture recognition seem to be very appealing technologies to achieve this goal.

Wearable gadgets like Google Glass is a promising example of ubiquitous computers that might be a an essential part of our lifestyle in the near future. Glass displays information in a smartphone-like hands-free format, that can communicate with the Internet via natural language voice commands. In this project, two new glass interaction techniques are proposed namely; eye tracking and vision-based gesture recognition.

Eye motion tracking serves as powerful tool to know the user's point of gaze and attentional state. Hence this information is used to increase the responsiveness of the computer in respond to users actions. Moreover eye-motions can be translated into commands for Glass. Another feature of eye tracking is that it can be used to provide aid to people with disabilities hindering them from casual interaction with wearable gadgets.

Gesture recognition has the potential to be a natural and powerful tool supporting efficient and intuitive interaction between the human and the computer (Glass). Visual interpretation of hand gestures can be interpreted into commands for Glass which definitely will help in achieving the ease and naturalness desired for Human Computer Interaction (HCI). Interpreting sign language is a very promising application of gesture recognition, offering an easy alternative way of interaction that will help many deaf people.



1. Introduction

Our project is based on two techniques, which are eye tracking and gestures recognition. Using these techniques, we can facilitate Human Computer Interaction. This will be very useful for people who have some disabilities. Handicapped who are unable to use their hands to touch the mouse and keyboard can interact with computers using their eye only. With eye tracking, we can detect the gaze point on the screen and determine where the person looks at in order to facilitate the handicapped communication.

The deaf people always use the sign language to interact with the things around them, Depending on the gesture recognition technique, It will be very useful for them to interact with the device using some gestures.

In general our application can make the interaction with the computer or any mobile device becomes easier, Even if the users can have no disabilities. It is very attractive if we use our eye to communicate with the screen or some signs with your hands to move through it, We will have no need for mouse or keyboard or any heavy devices.

1.1 Eye Tracking

1.1.1 Starburst

Starburst is a robust algorithm for video based eye tracking. We started with implementing Starburst Algorithm. This algorithm is more accurate than pure feature-based approaches yet is significantly less time consuming than pure model-based approaches. A validation study shows that the technique can reliably estimate eye position with an accuracy of approximately one degree of visual angle even in the presence of significant image noise.

The use of eye tracking has significant potential to enhance the quality of everyday human-computer interfaces. Two types of human-computer interfaces utilize eye-movement measures active and passive interfaces. Active interfaces allow users to explicitly control the interface through the use of eye movements. For example, eye-typing applications allow the user to look at keys on a virtual keyboard to type instead

of manually pressing keys as with a traditional keyboard. Similarly, systems have been designed that allow users to control the mouse pointer with their eyes in a way that can support, for example, the drawing of pictures. Active interfaces that allow users with movement disabilities to interact with computers may also be helpful for healthy users by speeding icon selection in graphical user interfaces or object selection in virtual reality. Passive interfaces, on the other hand, monitor the user's eye movements and use this information to adapt some aspect of the display. For example, in video transmission and virtual-reality applications, gaze contingent variable-resolution display techniques present a high level of detail at the point of gaze while sacrificing level of detail in the periphery where the absence of detail is not distracting. While eye tracking has been deployed in a number of research systems and to some smaller degree consumer products, eye tracking has not reached its full potential.

1.1.2 Robust Realtime Pupil Tracking

In this paper, they present a real-time dark-pupil tracking algorithm designed for low-cost head-mounted active-IR hardware. Their algorithm is robust to highly eccentric pupil ellipses and partial obstructions from eyelashes, making it suitable for use with cameras mounted close to the eye. It first computes a fast initial approximation of the pupil position, and performs a novel RANSAC based ellipse fitting to robustly refine this approximation.

1.1.3 Pupil

Pupil is an open source platform for pervasive eye tracking and mobile gazed interaction. In this paper, they argue that affordability does not necessarily align with accessibility. They define accessible eye tracking platforms to have the following qualities: open source components, modular hardware and software design, comprehensive documentation, user support, affordable price, and flexibility for future changes.

They have developed Pupil, a mobile eye tracking headset and an open source software framework, as an accessible, affordable, and extensible tool for pervasive eye tracking research. They explain the design motivation of the system, provide an in depth technical description of both hardware and software, and provide an analysis of accuracy and performance of the system.

2. Starburst Eye Tracking Algorithm

As proposed in the previous chapters that we rely on 2 different approaches to provide a new user experience; Eye-tracking, and gesture recognition. Through this chapter we will go through the details of our eye-tracking module, what algorithms we used, problems we faced, and workarounds to that problems.

2.1 Starburst

In the first version of our eye-tracking module we went for implementing the Starburst [9].The Starburst eye-tracking algorithm combines feature-based and model-based approaches to achieve a good trade off between run-time performance and accuracy for dark-pupil infrared illumination. The goal of the algorithm is to extract the location of the pupil center and the corneal reflection so as to relate the vector difference between these measures to coordinates in the scene image. As figure 2.1 shows, the algorithm begins by locating and removing the corneal reflection from the image. Then the pupil edge points are located using an iterative feature-based technique. An ellipse is fitted to a subset of the detected edge points using the Random Sample Consensus (RANSAC) paradigm. The best fitting parameters from this feature-based approach are then used to initialize a local model-based search for the ellipse parameters that maximize the fit to the image data.

2.2 Noise Reduction

The Starburst algorithms deals with two types of noise that are commonly encountered in any low cost head-mounted eye-tracker. The two types of noise are the shot noise and the line noise. Shot noise is reduced by applying a $5x5$ Gaussian filter with a standard deviation of 2 pixels. Line noise is reduced by applying a normalization factor is applied line by line shift to shift the mean intensity of the line to the running average derived from previous frames. The following modelling describes the process of line noise reduction as proposed in Startburst [9].

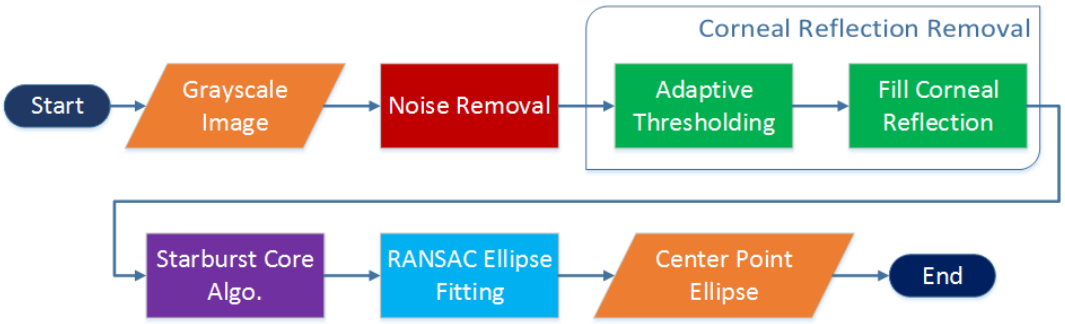


Figure 2.1: Overview of MIRT Algorithm

$$C(i,l) = \beta \bar{I}(i,l) + (1-\beta)C(i-1,l) \tag{2.1}$$

Where $C(i,l)$ is the normalization factor, $I(i,l)$ is the average line intensity and $B = 0.2$. Keep in consideration that the noise reduction step is optional and can be deactivated if the used camera is capable of capturing less noisy images.

In our implementation we didn't do any modifications to the noise reduction module. We implemented this module as described in the Starburst algorithm [9]. Note that using head-mounted camera in our system doesn't require noise reduction, so this module is deactivated at runtime.

2.3 Corneal Reflection

2.3.1 Detection and Localization

The corneal reflection corresponds to one of the brightest regions in the eye image. Thus the corneal reflection can be obtained through thresholding. However, a constant threshold across observers and even within observers is not optimal. Therefore we use an adaptive thresholding technique in each frame to localize the corneal reflection.

One of the heuristics adopted by the Starburst algorithm is that cornea extends approximately to the limbus, so the search space for the cornea can be limited with a square region of interest by half width of $h = 150$ which makes sense. Other heuristic which we believe is the core of the corneal reflection detection is that the corneal reflection is the largest and the brightest candidate region in the region of interest, as other specular reflections tend to be quite small and located off the cornea as well as near the corner of the image where the eye lids meet.

At first the input (gray scale) frame is converted to binary form, only values above the maximum threshold are taken as corneal reflection candidates. Then the ratio between the area of the largest candidate to the average area of other regions is calculated as the threshold is lowered. A notable observation is that the intensity of the corneal reflection monotonically decreases towards the edges.

At the beginning the ratio between largest candidate area and average area of other candidates will increase since the corneal reflection will grow in size faster than other

areas. As the threshold is decreased the ratio will begin to drop because the false candidates are becoming of significant size compared to the corneal reflection. The optimal threshold is the threshold that generates the hight ratio. The largest candidate using the optimal threshold is considered the corneal reflection with the geometric center (c_x, c_y).

In the Starburst algorithm the corneal reflection intensity profile is assumed to follow a bivariate Gaussian distribution. Full extent of the corneal reflection is obtained by relating the the radius r where the average decline in intensity is maximal to the radius with maximal decline for a Gaussian (i.e. a radius of one standard deviation). To capture 99% of the corneal reflection profile use $2.5r$. Where r is computed through a gradient decent search that minimizes:

$$\frac{\int I(r+\delta, x_c, y_c, \theta) d\theta}{\int I(r-\delta, x_c, y_c, \theta) d\theta} \tag{2.2}$$

where $\delta = 1$ and $I(r, x, y, \theta)$ is the pixel intensity at angle θ on the contour of a circle defined by the parameters r, x, and y. The search is initialized with $r = \sqrt{area/\pi}$, where area is the number of pixels in the thresholded region. The search converges rapidly.

In our implementation we used a simplified version for corneal reflection detection procedure described by the Starburst algorithm. First step we convert the input gray scale image to binary form. Then we perform adaptive thresholding until we find the optimal threshold (which generates the highest ratio) as described in the original algorithm. Next we estimate the the center of corneal reflection region. Till now no differences from the original algorithm. The main difference is that we assume that the corneal reflection is circular and hence we find the radius from the area of circle relation $r = \sqrt{area_{max}/\pi}$.

2.3.2 Removal

Original Starburst algorithm uses radial interpolation to remove the corneal reflection. First, the central pixel of the identified corneal reflection region is set to the average of the intensities along the contour of the region. Then for each pixel between the center and the contour, the pixel intensity is determined via linear interpolation.

Our implementation of the corneal reflection removal is different to the original algorithm, we assume that the corneal reflection is circular and hence we fill a circular region in the image with an estimation of the pupil intensity. The pupil intensity is estimated by averaging the intensity along the ring that encapsulates the corneal reflection and is d pixels larger than corneal reflection radius, where d is set to 10 pixels.

2.4 Starburst Algorithm

Unlike most of feature based eye tracking approaches that apply edge detection to the entire image, Starburst algorithm detects edges along a limited number of rays that extend from a central best guess of the pupil center. These rays can be seen in Figure **??**. This method takes advantage of the high-contrast elliptical profile of the pupil contour

present in images taken using the dark-pupil technique. Algorithm pseudo code is shown in algorithm 1.

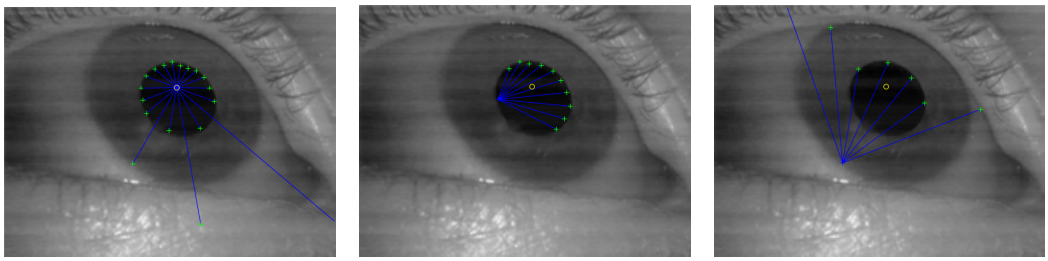
Algorithm 1 Starburst Original Algorithm

Input: Eye image - corneal reflection removed, Best guess of pupil center
Output: Set of feature points

```
1:  procedure STARBURST
2:      repeat
3:          Stage 1:
4:          Follow rays extending from the starting point
5:          Calculate intensity derivative at each point
6:          if derivative > threshold then
7:              Place feature point
8:              Halt marching along ray
9:          Stage 2:
10:         for all feature points detected in Stage 1 do
11:             March along rays returning towards the start point
12:             Calculate intensity derivative at each point
13:             if derivative > threshold then
14:                 Place feature point
15:                 Halt marching along ray
16:         Starting point = geometric center of feature points
17:     until starting point converges
```

For each frame a location is chosen that represent the best guess of the pupil center in this frame. The best guess at the first frame is chosen to be the center of the image, best guess at frame i are set to be the center of the pupil detected in the previous frame (frame $i-1$). Next, the derivatives $\triangle$ along the $N = 18$ rays, extending radially away from this starting point, are independently evaluated pixel by pixel until a threshold $\phi = 20$ is exceeded. Since Starburst uses a dark-pupil technique, only positive derivatives (increasing intensity as the ray extends) are considered. When the value of the derivative along the line exceeds the threshold a features point (pupil edge candidate point) is marked, and the processing along this line is halted. If the processing along a line reached borders of the image no feature point is marked. Example of the candidate feature points and corresponding rays are shown at figure 2.2.

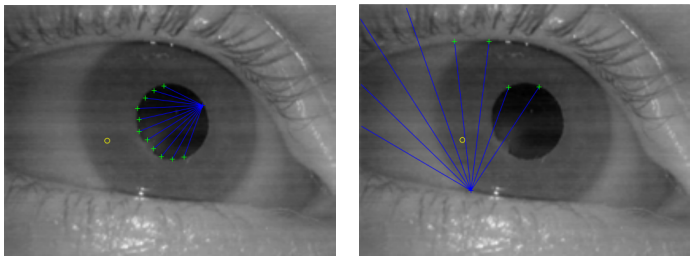
For every feature point obtained the above described algorithm is repeated with a minor modification. Search rays are limited to $\gamma = \pm 50$ degrees from the original ray that originally produced the feature point. This procedure tends to increase the ratio of the number of feature points on the pupil contour over the number of feature points not on the pupil contour. If the pivot feature point lies on the pupil contour then the rays around the original ray will result in feature points that also lie on the opposite half



(a) Stage 1 rays (b) Stage 2 inlier point (c) Stage 2 outlier point

Figure 2.2: Visualization of Starburst stage 1 and stage 2 feature point detection procedure

of the pupil contour. On the other hand if the original feature point is not on the pupil contour, then limiting the search space will cut down the number of feature point that doesn't lie on the pupil contour. One important note to take into consideration is that the number of rays in this step is only 10 (5 on each side of the original ray). Figure 2.3 shows the two cases described above.



(a) Stage 2 inlier point (b) Stage 2 outlier point

Figure 2.3: Visualization of Starburst stage 2 feature point detection procedure

Using the two stage feature detection process improves the robustness of the method to poor initial guesses for the starting point. However, the feature points tend to bias to the side of the pupil contour nearest the initialization point. Although another iteration of the ray process would minimize this bias, the computational burden grows exponentially with each iteration and thus would be an inefficient strategy.

Fitting an ellipse to the set of biased feature points will induce significant error into fit. A good strategy to eliminate the bias is to iterate the recently described procedure, with each time computing the average of the detected feature points to be the new estimated center of the pupil for the next iteration. The iterating process is halted when the estimated pupil center between two successive iteration is less than $d = 10$ pixels. Starburst paper mention that if guess is a good estimate of the pupil center, only a single iteration is required. When the initial estimate is not good, typically only a few iterations (< 5) are required for convergence. If convergence is not reached within $i = 10$iterations, as occurs sometimes during a blink when no pupil is visible, the algorithm halts and begins processing the next frame.

In out implementation of this module we sticked to the exact Startburst algorithm described in the paper.

2.5 Ellipse Fitting

Ellipse Fitting Given a set of feature points, the function of this module is to find best fitting ellipse. Most of algorithms use least-squares ellipse fitting including all feature points (e.g. see [14]). Using all points to estimate an ellipse results in significant error that strongly affect the accuracy. Notice that a few feature points not on the pupil contour dramatically reduces the quality of the fit to an unacceptable level.

To address this issue Starburst uses Random Sample Consensus (RANSAC) paradigm for model fitting [3]. Starburst paper claims that Starburst is the first algorithm that uses RANSAC in the context of eye-tracking. RANSAC is an effective technique for model fitting in the presence of a large but unknown percentage of outliers in a measurement sample. A basic assumption is that the data consists of "inliers", i.e., data whose distribution can be explained by some set of model parameters, though may be subject to noise, and "outliers" which are data that do not fit the model. In our application inliers are feature points that lie on the pupil contour, while outliers are feature points that belong to other contours other than the pupil contour. Least-squares approaches assumes that all sample points fit the model and hence, uses all the sample points to estimate the model. On the other hand RANSAC is an iterative method to estimate parameters of a mathematical model from a set of observed data which contains outliers. General outline of RANSAC technique is shown at algorithm 2.

Algorithm 2 General RANSAC Procedure

```
1: procedure RANSAC
2:     while i < N do
3:         Draw s points uniformly at random
4:         Fit model to these s points
5:         Find inliers to this model among the remaining points (points whose error
   < t)
6:         If there are d or more inliers, accept the model and refit using all inliers
```

The output of the two stage feature detection process may result in very few outliers in some cases, while in other cases outlier prevail. Using the RANSAC technique increase the ability of the system to do robust estimation of the model (ellipse) parameters. The following procedure is repeated R times. First, five samples are randomly drawn from the set of feature set obtained by the previous module. Singular Value Decomposition (SVD) on the conic constraint matrix generated with normalized feature-point coordinates [7] is used to find the parameters of the ellipse that perfectly fit these five points. Then, the number of candidate feature points in the data set that agree with this model (i.e. the inliers) are counted. Inliers are those sample points for which the algebraic distance to the ellipse is less than some threshold t.

Our implementation of this stage differs a bit from the implementation of the original Starburst. We draw 5 samples from the detected feature set, then we compute the best fit ellipse using Fitzgibbon's algorithm [4] which is implemented in OpenCV. Finally we evaluate the error between the estimated model and all feature points and count inliers/outliers. Algorithm 3 summarizes how we use RANSAC in our system.

Algorithm 3 Our RANSAC Procedure

```
1: procedure RANSAC
2:     while i < N do
3:         Draw s points uniformly at random
4:         Fit ellipse model to these s points
5:         Find inliers to this ellipse (points whose algebraic error < t)
6:         If there are d or more inliers, accept the ellipse and refit using all inliers
```

As a matter of fact we tried different approaches in our implementation, at first we tried to implement the same approach mentioned in the Starburst paper. However finding the SVD of the parameter matrix using OpenCV SVD didn't produce consistent results because the calculated eigen vectors/values were not correct. Some books like [1] addressed this issue and suggested using Eigen mathematics library [5] to find the eigen values/vector. In our application we couldn't benefit from this solution since, Eigen library is only available in C++ while the user end if our application is in Java (Android user end).

2.5.1 Estimation of Number of Iterations R

Starbusrt algorithm assumes that the average error variance of the feature detector is approximately one pixel and that this error is distributed as a Gaussian with zero mean. Thus to obtain a 95% probability that a sample is correctly classified as an inlier, the threshold should be derived from a χ^2 distribution with one degree of freedom [7]. This results in a threshold distance of $t = 1.98$ pixels.

The parameter R (the number of iterations), however, can be determined from a theoretical result. Let p be the probability that the RANSAC algorithm in some iteration selects only inliers from the input data set when it chooses the s points from which the model parameters are estimated. When this happens, the resulting model is likely to be useful so p gives the probability that the algorithm produces a useful result. Let w be the probability of choosing an inlier each time a single point is selected, that is,

$$w = \frac{number\,of\,inliers\,in\,data}{number\,of\,points\,in\,data}$$

it can be proven that

$$R = \frac{log(1-p)}{log(1-w^5)} \tag{2.3}$$

2.6 Homographic Mapping and Calibration

In order to calculate the point of gaze of the user in the scene image, a mapping between locations in the scene image and an eye-position measure must be determined. The typical way to determine the mapping is via a calibration process. During calibration, the user is required to look at a number of scene points for which the positions in the scene image are known. At each position the eye-position $\vec{e} = (x_e, y_e, 1)$ and the scene position $\vec{s} = (x_s, y_s, 1)$ is measured. The mapping is generated using the $3x3$ Homography matrix which has 8 degrees of freedoms. Each point (map from scene to eye) produces 2 independent equations. Hence four mapping points are needed to compute the entries of the homography matrix up to scale [10].

In our implementation we use the OpenCV *findHomography* procedure to find the homography matrix. Given a set of points in image coordinates and corresponding set of points in the eye coordinates, *findHomography* finds a perspective transformation between two coordinate systems.

Once this mapping is determined the user's point of gaze in the scene for any frame can be established as $\vec{s} = H\vec{e}$.

3. Pupil Eye Tracking Algorithm

As proposed in the previous chapters that we rely on Starburst [9] Algorithm to get the center of the pupil. But we found that Starburst is not robust and accurate, so we will use Pupil Algorithm to enhance eye-tracking module. Through this chapter we will go through the details of our eye-tracking module by using Pupil Algorithm.

3.1 Pupil

In the second version of our eye-tracking module we went for implementing the Pupil Algorithm [8]. Pupil eye-tracking algorithm is an accessible, affordable and extensible tool for pervasive eye tracking research. This chapter we will explain the design motivation of the algorithm, provide an in depth technical description of algorithm. We will use a wearable mobile eye tracking headset with one scene camera and one infra-red (IR) spectrum eye camera for dark pupil detection. Both cameras connect to a laptop, desktop, or mobile computer platform via high speed USB 2.0. The camera video streams are read for real-time pupil detection, gaze mapping and recording.

3.2 System Design Objectives

Pupil leverages the rapid development cycle and scaling effects of consumer electronics, USB cameras and consumer computing hardware, instead of using custom cameras and computing solutions.

Pupil algorithm is open source and strives to build and support a community of eye tracking researchers and developers.

3.3 Cameras

The scene camera mount and eye camera mount interface geometries are open source. By releasing the mount geometry we automatically document the interface, allowing users to develop their own mounts for cameras of their choice.

Pupil uses USB interface digital cameras that comply with the UVC (USB Video Class) standard. The Pupil headset can be used with other software that supports the UVC interface. Pupil can be easily extended to use two eye cameras for binocular set-ups and more scene cameras as desired.

3.3.1 Eye Camera

We use a small and lightweight eye camera to reduce the amount of visual obstruction for the user and keep the headset lightweight. The eye camera can capture at a maximum resolution of 800x600 pixels at 30 Hz. Using an IR mirror ("hot mirror") was considered as strategy to further reduce visual obstruction.

Pupil uses the "dark pupil" detection method. This Requires the eye camera to capture video within a specific range of the IR spectrum.

3.3.2 Scene Camera

The scene camera is mounted above the user's eye aligning the scene camera optics with the user's eye along a sagittal plane. The scene camera faces outwards to capture a video stream of a portion of the users FOV at 30Hz. The scene camera lens has a 90 degree diagonal FOV. The scene camera is not only high resolution (max resolution 1920x1080 pixels), but also uses a high quality image sensor. This is very advantageous for further computer vision and related tasks performed in software.

3.4 Computing Device

The Pupil eye tracking algorithm works in conjunction with standard multi-purpose computers: laptop, desktop, or tablet. Designing for user supplied recording and processing hardware introduces a source for compatibility issues and requires more set-up effort for both users and developers. However, enabling the user to pair the headset with their own computing platform makes Pupil a multi-purpose eye tracking and analysis tool. Pupil is deployable for lightweight mobile use as well as more specialized applications like: streaming over networks, geotagging, multi-user synchronization; and computationally intensive applications like real time 3D reconstruction and localization.

3.5 Pupil Detection Algorithm

The pupil detection algorithm locates the dark pupil in the IR illuminated eye camera image. The algorithm does not depend on the corneal reflection, and works with users who wear contact lenses and eyeglasses.

The pupil detection algorithm is under constant improvement based on feedback collected through user submitted eye camera videos. Here we provide a description of default pupil detection algorithm.

3.5.1 overview

The eye camera image is converted to gray-scale. The initial region estimation of the pupil is found via the strongest response for a center-surround feature as proposed by Swirski et al. [12] within the image.

Detect edges using Canny [2] to find contours in eye image. Filter edges based on neighbouring pixel intensity. Look for darker areas (blue region). Dark is specified using a user set offset of the lowest spike in the histogram of pixel intensities in the eye image.

Filter remaining edges to exclude those stemming from spectral reflections. Remaining edges are extracted into into contours using connected components [11]. Contours are filtered and split into sub-contours based on criteria of curvature continuity.

Candidate pupil ellipses are formed using ellipse fitting [4] onto a subset of the contours looking for good fits in a least square sense, major radii within a user defined range, and a few additional criteria.

The results are evaluated based on the ellipse fit of the supporting edges and the ratio of supporting edge length and ellipse circumference (using Ramanujans second approximation [6]). We call this ratio "confidence". If the best results confidence is above a threshold the algorithm reports this candidate ellipse as the ellipse defining the contour of the pupil. Otherwise the algorithm reports that no pupil was found.

Figure 3.2 shows a performance comparison between Pupil's pupil detection algorithm, the stock algorithm proposed by Swirski et al., the ITU gaze tracker and Starburst on the benchmark dataset by Swirski et al. [12]. As error measure we used the Hausdorff distance between the detected and hand-labeled pupil ellipses [12]. We additionally conducted a test excluding the dataset p1-right, that contains eye images recorded at the most extreme angles. As can be seen from the Figure, Pupil without p1-right compares favourably to all other approaches. With an error threshold of 2 pixels Pupil achieves a detection rate of 80%; at 5 pixels error detection rate increases to 90%

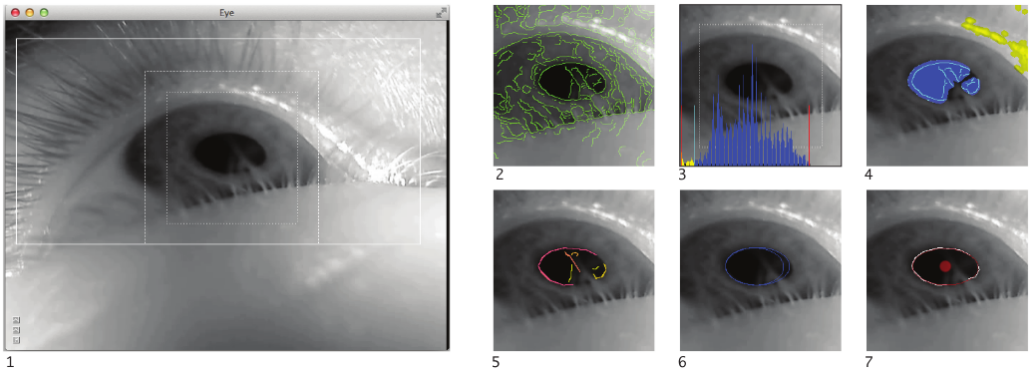
3.5.2 Integral Image

Rectangle features can be computed very rapidly using an intermediate representation for the image which we call the integral image. The integral image at location x,y contains the sum of the pixels above and to the left of x,y, inclusive:

$$ii(x,y) = \sum_{x'<=x,y'<=y} i(x',y'),$$

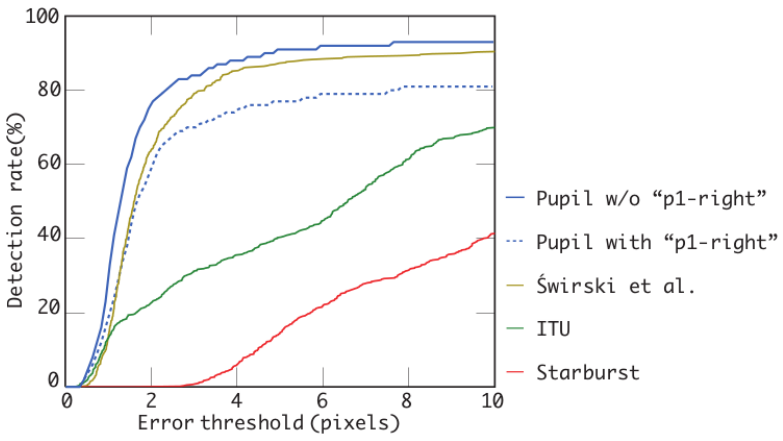
where $ii(x,y)$ is the integral image and $i(x,y)$ is the original image. Using the following pair of recurrences:

$$s(x,y) = s(x,y-1) + i(x,y) \tag{3.1}$$



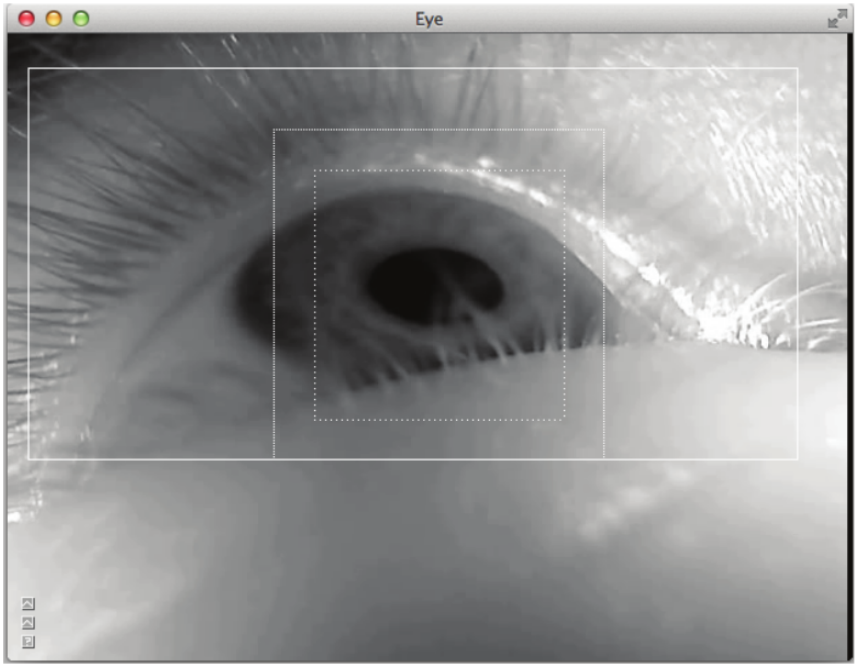
(a) Algorithm Overview

Figure 3.1: Vizualization of pupil detection algorithm. 1) Eye image converted to gray scale, user region of interest (white stroke rectangle), and initial estimation of pupil region (white square and dashed line square.) 2) Canny edge detection (green lines). 3) Define "dark" region as offset from lowest spike in histogram within eye image. 4) Filter edges to exclude spectral reflections(yellow) and not inside "dark" areas (blue). 5) Remaining edges extracted into contours using connected components and split into sub-contours based on curvature criteria (multi colored lines). 6) Candidate pupil ellipses (blue) are formed using ellipse fitting. 7) Final Ellipse fit found through an augmented combinatorial search(finally ellipse with center in red)-supporting edge pixels drawn in white.



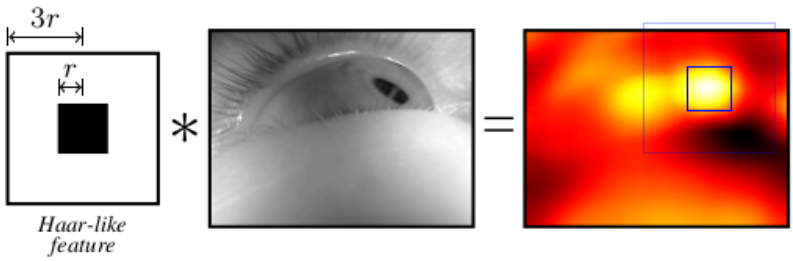
(a) Comparison

Figure 3.2: Comparison of pupil detection rate for Pupil’s algorithm, the stock algorithm proposed by Swirski et al., the ITU gaze tracker and Starburst.



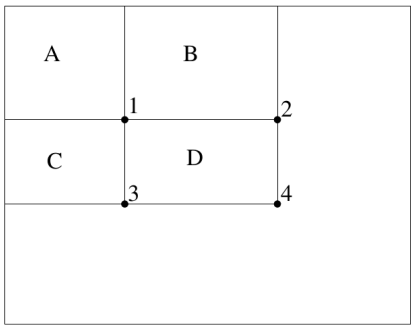
(a) Initial Estimation of Pupil Region

Figure 3.3: User region of interest (white stroke rectangle), and initial estimation of pupil region (white square and dashed line square.)



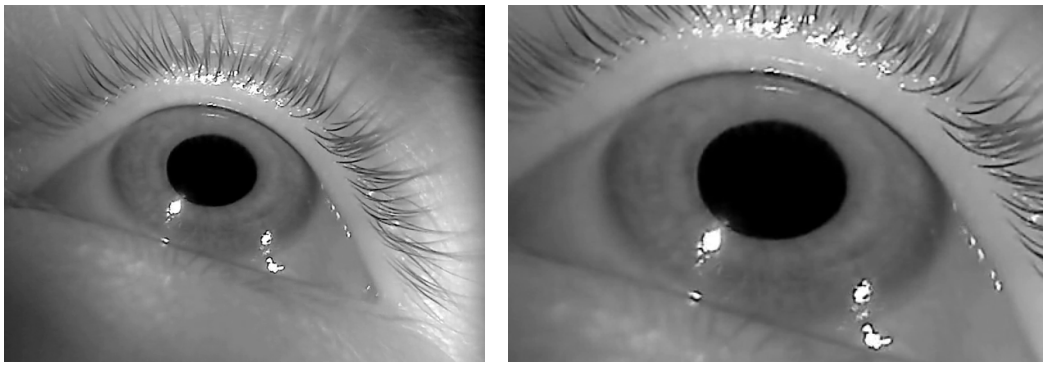
(a) Estimation of Pupil Region

Figure 3.4: To find the approximate pupil region, the eye image is convolved with a Haar-like centre surround feature of radius r. The pupil region is centred on the location of the maximal response over all pixels and radii



(a)

Figure 3.5: To find the approximate pupil region, the eye image is convolved with a Haar-like centre surround feature of radius r. The pupil region is centred on the location of the maximal response over all pixels and radii



(a) Original Eye Image

(b) Region of interest(ROI) after Running Initial Estimation Of Pupil Module

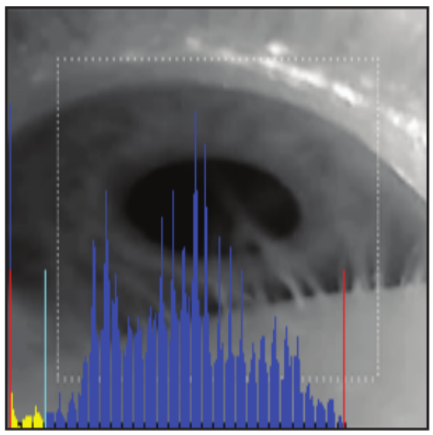
Figure 3.6: Input & Output of Initial Estimation of Pupil Module.

$$\text{ii(x,y)} = \text{ii(x-1,y)} + \text{s(x,y)} \tag{3.2}$$

(where $s(x,y)$is the cumulative row sum, $s(x,-1) = 0$, and $ii(-1,y) == 0$) the integral image can be computed in one pass over the original image.
Using the integral image any rectangular sum can be computed in four array references figure 3.5. Clearly the difference between two rectangular sums can be computed in eight references. Since the two-rectangle features defined above involve adjacent rectangular sums they can be computed in six array references, eight in the case of the three-rectangle features, and nine for four-rectangle features.[13]

3.5.3 Initial Estimation Of Pupil

Our initial region estimation assumes that the pupil region, either the dark pupil itself or the combination of pupil and iris, can roughly be described as a " dark blob surrounded by a light background ", and is the strongest such feature in the image. To find the pupil,



(a) Eye Image after Computing Histogram

Figure 3.7: After Computing Histogram to get Spikes.

we use a Haar-like feature detector, similar to the features used in cascade classifiers [13].

The core idea of the feature detector can be explained in terms of convolution. To find possible pupil regions, we convolve the image with a Haar-like centre-surround feature of a given radius figure 3.4 . We repeat this for a set of possible radii, between a user specified minimum and maximum, and find the strongest response over the 3D space of (x, y) and radii. The (x, y) location of this strongest response is assumed to be the centre of the pupil region, with the size of the region determined by the corresponding radius.

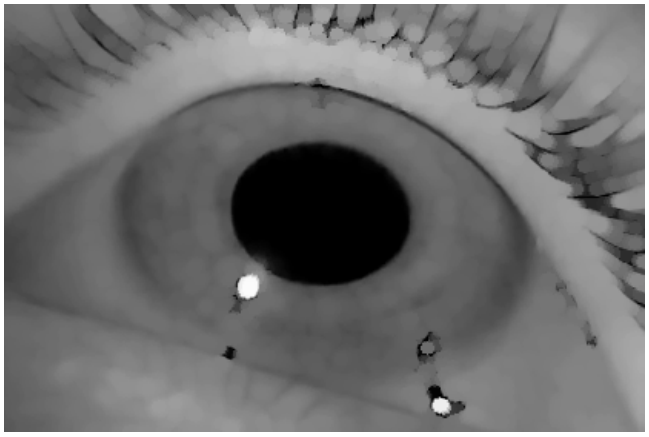
Although such a convolution is a slow operation if performed naively, we optimise this by first calculating the integral image [13]. Using this integral image, we can find the response of a pixel to a Haar-like feature in constant time, only needing to sample 8 pixel values, one for each corner of the two squares, thereby making this step linear in the number of pixels and possible radii.
Input and Output of this module is shown in Figure 3.6

3.5.4 Getting Dark Region(Pupil Region)

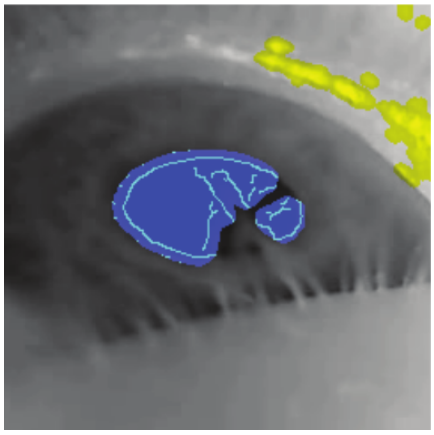
Here we want to extract the region which include the pupil by getting the most dark region in eye image. And then we filter the eye image by applying some Morphological operations like Dilation and Erosion to remove the noise to get a pure dark region.

At first we should get the histogram of the gray image of Region Of Interest (ROI) which we get from Initial Estimation Of Pupil Module. Then we should deduce spikes from histograms. After getting the lowest and highest spikes in histogram we can get the binary image of ROI by using lowest spike + user offset(usually we set offset by 11) as threshold as shown in figure 3.7.

Then, we should do some Morphological operation on binary image. to filter the image from any noise and make contours more clearer, so we apply Erosion at first and then Dilation(Closing Operation) as shown in Figure 3.8.

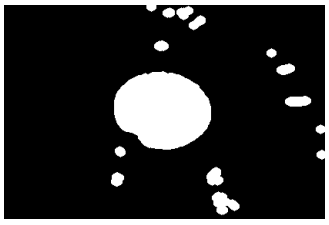


(a) Eye image after applying Closing operation on Eye image.



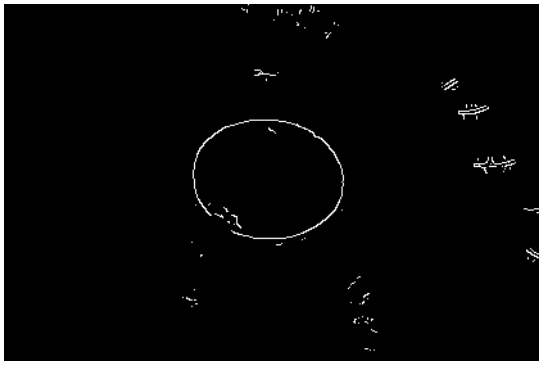
(b) Dark region is indicated by the Blue Region

Figure 3.8: After Applying Morphological operations(Closing operations).



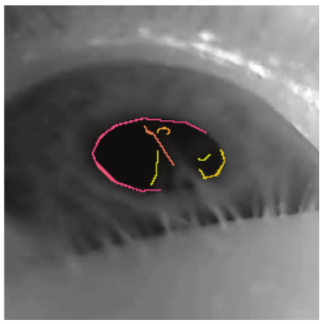
(a) Eye image after applying Closing operation on Eye image.

Figure 3.9: Final Result of Getting Dark Region Module.



(a) Edges of eye image after Excluding Spectral Reflections

Figure 3.10: Final Result of Excluding Spectral Reflection module.



(a) Multi coloured lines represent different contours

Figure 3.11: Final Result of Extracting Contour module.

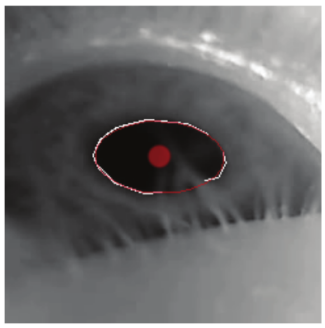
Finally we get the ROI as a binary image include only the Dark Region which represents Pupil as shown in Figure 3.9.

3.5.5 Excluding Spectral Reflection

We need to filter all the edges which result from Canny edge detector, by excluding the spectral reflection which appears as yellow lines in figure 3.8(b) that is out of the dark region. So, we need to remove edges in areas which are not dark enough and where the glints is (Spectral Reflection from IR leds).

We get minimum edges from comparing edges image which result from Canny edge detector with the Spec Mask (Mask that we get from the in range thresholding with highest spike and 255) and then we reduce the number of edges by comparing the edges image with binary Mask(Mask which we get from in range thresholding with lowest spike with 255).

At the end we remove the Spectral reflection and the binary eye image is filtered from any noise as shown in Figure 3.10.



(a) Final ellipse with center in red and supporting edge pixels are drawn in white.

Figure 3.12: Final Result of Ellipse Fitting module.

3.5.6 Extracting Contours

After getting the minimum edges we can extract Contours by using connected components, and then split the contours into sub-contours based on curvature continuity criteria as shown in Figure 3.11. [11] We have used OpenCV findCountours method to get all

the contours from edges and then we get the approximate Poly-lines that each contour forms. So that, we can use each poly-line to get the angle between each subsequent 2 points to make sure that this contour is a good contour and also it helps to split contours into sub-contours based on angles between two subsequent points.

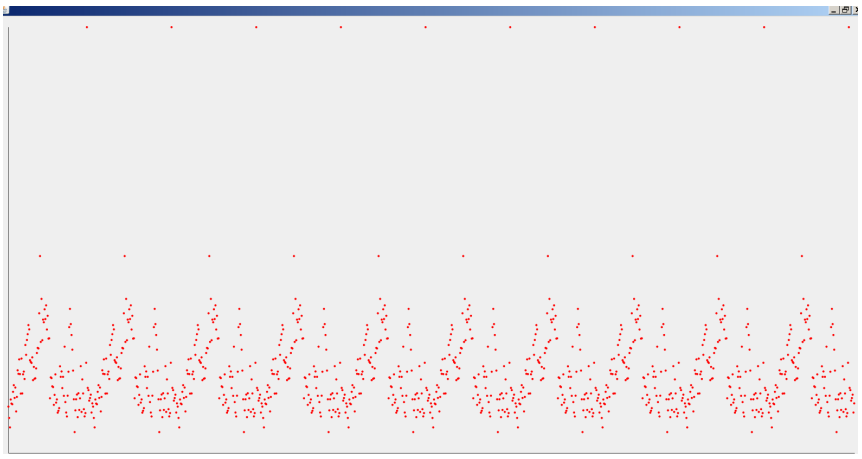
3.5.7 Ellipse Fitting

Finally we should estimate the best ellipse that surrounds the Pupil by fitting an ellipse found through an augmented combinatorial search. [4]

Candidate pupil ellipses are formed using ellipse fitting onto a subset of the contours looking for good fits in a least square sense, major radii within a user defined range and a few additional criteria. An augmented combinatorial search looks for contours that can be added as support to the candidate ellipses. The results are evaluated based on the ellipse fit of the supporting edges and the ratio of supporting edge length and ellipse circumference. We call this ratio " confidence ". If the best results confidence is above a threshold the algorithm reports this candidate ellipse as the ellipse defining the contour of the pupil as shown in Figure 3.12. Otherwise the algorithm reports that no pupil was found.

3.6 Performance Evaluation

We have used Data Set of Computer Laboratory, University of Cambridge, United Kingdom which is used at Robust real-time pupil tracking in highly off-axis images to test performance. [12]
These Data Set contains video frames as individual images, and a text file describing



(a)

Figure 3.13: Error graph for Pupil algorithm performance test

the pupil ellipse in selected frames. The format of each line in text file is <frame #> $|<x><y><a><b><\theta>$, such that x,y represents the pupil center in frame# , a,b represents the ellipse major radius and minor radius values and θ represents angle between the major axis and the x-axis in radians. Total number of frames is 940 frame. Finally after Testing our implementation on this Data Set, we have make a comparison between our results and the original pupil center location. So we found the Average error = 1.545478. You Can see the Error graph in Figure 3.13.

4. MIRT

In this chapter we present MIRT: Morpholgical Based Iris Tracking a novel method for real-time iris detection and tracking. MIRT is a morphological based real-time iris detection algorithm. Morphological operations are simple and fast to compute which reduces the time required to detect the iris.

Two typed of image processing in eye tracking are used; visible and infrared spectrum imaging. Most available eye tracking approaches like Starburst [9] and Pupil [8] rely on infrared spectrum imaging. It's known that visible spectrum imaging is more complicated due to the uncontrolled ambient light that is used as a light source which usually contains multiple specular and diffuse components. On the other hand infrared imaging eliminates uncontrolled specular reflection by illuminating the eye with a uniform and controlled infrared light [9]. However, infrared imaging requires using special hardware (infrared cameras, infrared lights). MIRT uses visible imaging to track the iris. Not to mention that MIRT can be used outdoors which is not available using infrared based eye-tracking algorithms We also doesn't require any special hardware, we use a low resolution webcam. Used hardware and head-mounted tracker will be discussed later in this book.

4.1 Overview

MIRT algorithm can be divided into 2 stages; 1) locating region of interest in the given image, and 2) detecting the iris in the region of interest located in stage 1. At first we transform the input frame to gray scale then we locate the parts of the visible parts of the sclera. Next we estimate the location of the iris based on the location of the sclera, and we crop the region of interest (ROI) from the source image. We try to minimize the ROI as possible to cut down the runtime and to minimize the error of the estimated iris center. From the extracted ROI we find the a set of feature points. An inlier feature points is a point that lies on the edge of the iris. Extracted feature points may contain some outliers, therefore we use RANSAC ellipse fitting to fit ellipse model on the extracted feature points. RANSAC is an effective technique for model fitting in the presence of a large but unknown percentage of outliers in a measurement sample. Using the RANSAC

technique increase the ability of the system to do robust estimation of the model (ellipse) parameters. Figure 4.1 shows the stages of the MIRT algorithm.

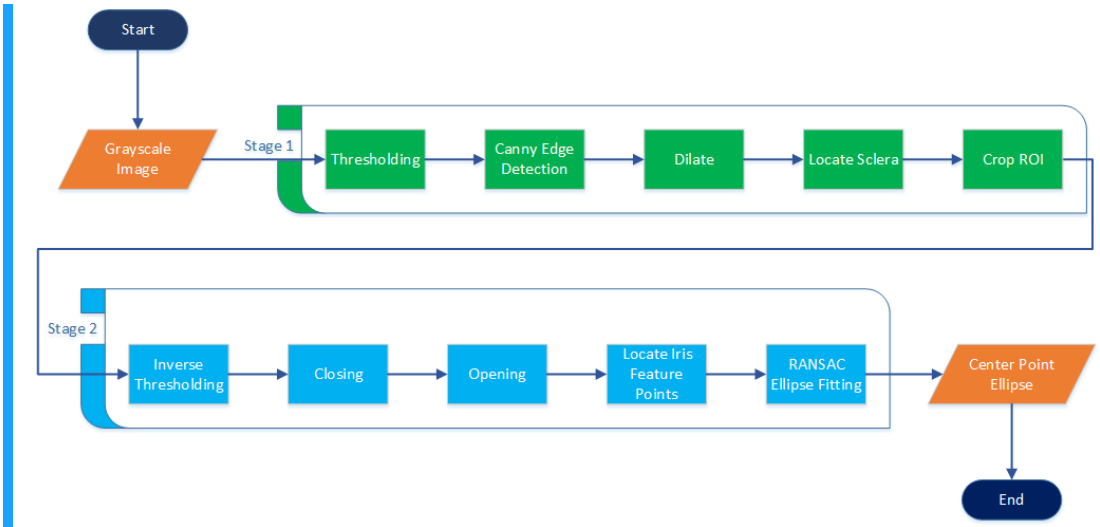


Figure 4.1: Overview of MIRT Algorithm

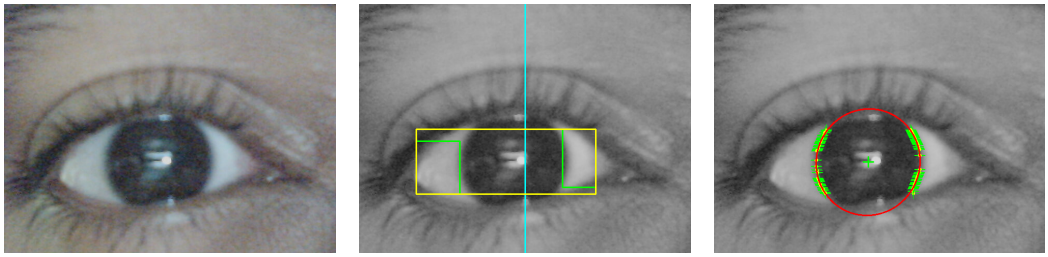
4.2 ROI Localization

As figure 4.1 illustrates the pipeline of the ROI localization step. We use simple thresholding on the input gray scale image (frame). Then we apply Canny edge detection algorithm [2] on the binary image. Then we dilate using a small structuring element just to fill the small gaps between edges and make edges wider. This goal from the dilation step is to increase the robustness of contour finding step. We use circular shaped small structuring element with diameter $d = 3$ pixels.

The next stage is that we find the largest 2 contours that satisfy the size constrain. We defined a size threshold to eliminate the regions that might be mistaken with the sclera. When the iris is in the middle of limbus (corneal limbus is the border of the cornea and the sclera) 2 parts of the sclera are visible as illustrated in figure 4.2(b). In this case the largest 2 contours will be two parts of the sclera and the iris is estimated to lie between those two parts. The ROI is set to be the bounding box of the two contours, shown in figure 4.2(b).

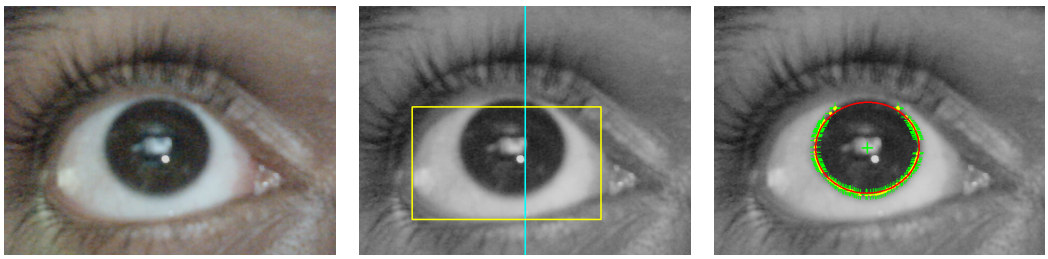
In other cases when the iris is either on the left or on the right of the limbus, only one part of the sclera is visible which is the largest contour found in the image, hence we have 3 different situations. If the width of the largest contour is greater than twice the estimated iris size $h = 150$ (based on our head mounted camera) then the ROI is the bounding rectangle of the largest contour as illustrated in figure 4.3. We assume that the iris is nearly at the center of the limbus, yet one of the visible sclera contours couldn't be extracted.

For the two other cases we define a placement ratio



(a) Input image (b) Illustration of stage 1 (c) Algorithm output

Figure 4.2: MIRT in action when 2 parts of sclera are visible; (a) shows the input image. (b) shows the 2 parts of sclera marked with green, ROI marked with yellow, and previous estimation of pupil center x with cyan. (c) shows the estimated ellipse with red and feature points with green crosshairs.



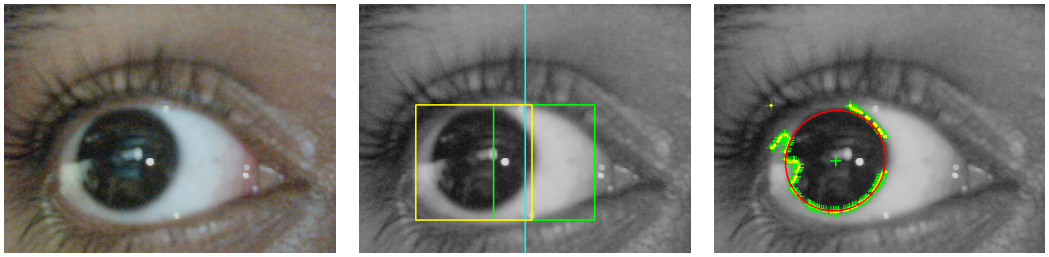
(a) Input image (b) Illustration of stage 1 (c) Algorithm output

Figure 4.3: MIRT in action when 2 connected parts of sclera are visible; (a) shows the input image. (b) shows the visible contour of sclera marked with green, ROI marked with yellow, and previous estimation of pupil center x with cyan. (c) shows the estimated ellipse with red and feature points with green crosshairs.

$$r = abs(\frac{c - x_{start}}{x_{end} - c})$$

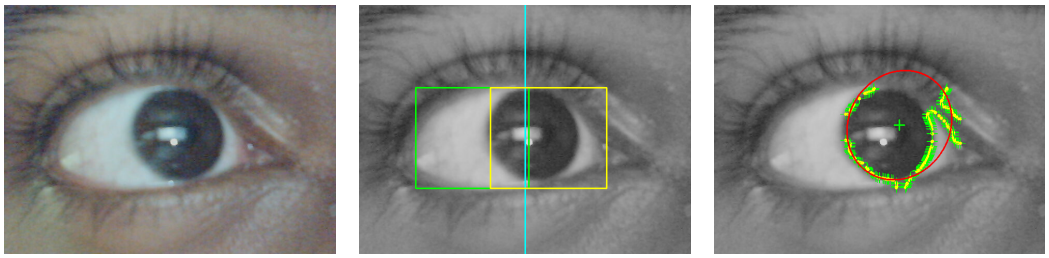
where c is the x coordinate value of the iris center estimated in the previous frame, c is set to the center of the image in the first frame. x_{start} and x_{end} are the x coordinates of the start and the end of largest contour. In other words r is the ratio between the both sides of the largest contours to the initial center of the pupil. If $r \leq 1$ then the iris is on the right and the left part of the sclera is visible. The ROI width stretches from $x_{end} - h/2$ to $x_{end} + h$ and have the same hight of the largest. See figure 4.4 for illustration.

The opposite case is that if $r > 1$ then the iris is in the left and the right part of the sclera is visible. Similar to the previous case the ROI is set to be $x_{start} - h$ to $x_{start} + h/2$ in width and also have the same height of the largest contour. Figure 4.5 shows this case. One last case remaining is that none of the found contours satisfied the size condition, this situation can occur in frames where the limbus is not visible (i.e blink), hence the frame is skipped. Finally if the estimated bounds of the ROI are outside input image boundaries the ROI boundaries are clamped to the input image bounds. Algorithm 4 illustrates the procedure of MIRT ROI localization stage.



(a) Input image (b) Illustration of stage 1 (c) Algorithm output

Figure 4.4: MIRT in action when the right side of sclera is visible; (a) shows the input image. (b) shows the visible (right) side of sclera marked with green, ROI marked with yellow, and previous estimation of pupil center x with cyan. (c) shows the estimated ellipse with red and feature points with green crosshairs.



(a) Input image (b) Illustration of stage 1 (c) Algorithm output

Figure 4.5: MIRT in action when the left side of sclera is visible; (a) shows the input image. (b) shows the visible (left) side of sclera marked with green, ROI marked with yellow, and previous estimation of pupil center x with cyan. (c) shows the estimated ellipse with red and feature points with green crosshairs.

4.3 Locate Iris In ROI

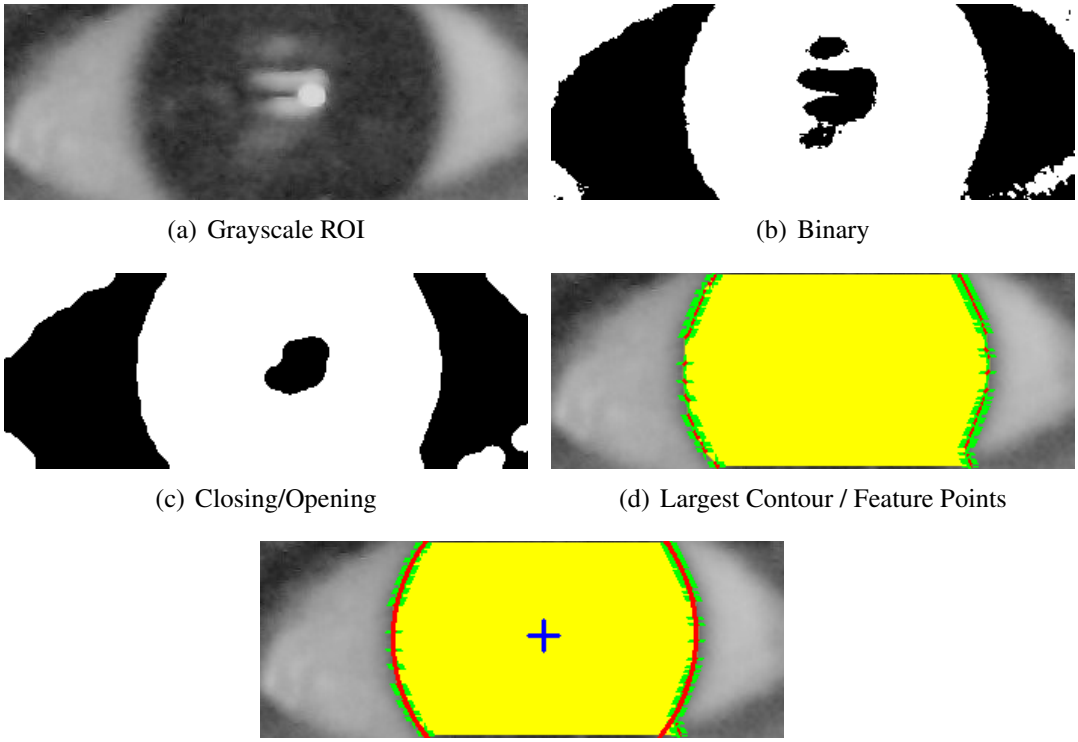
In this stage we are interested in finding the iris in the tightly cropped ROI located in stage 1. we start by applying inverse simple thresholding to filter dark areas in the ROI. Next we apply morphological closing followed by opening using 19 pixel circular structuring element. The main goal of this step to fill the gaps in the iris (gaps come from light reflections) and to open between the iris and other parts of the ROI that passed through the thresholding step. Then we find the largest contour in the ROI. In almost all cases the largest contour will contain the iris as shown in figure 4.6(d). Finally we extract feature points from the largest contour. Edge points of the contour are considered as feature points. We use Random Sample Consensus (RANSAC) paradigm for model fitting [3]. RANSAC is an effective technique for model fitting in the presence of a large but unknown percentage of outliers in a measurement sample. In our application inliers are feature points that lie on the iris contour, while outliers are feature points that belong to other contours other than the iris contour. Using the RANSAC technique increase the ability of the system to do robust estimation of the model (ellipse) parameters. We draw 5 samples from the detected feature set, then we compute the best fit ellipse using Fitzgibbon's algorithm [4] which is implemented in OpenCV. Finally we evaluate

Algorithm 4 MIRT: Locate ROI (Stage 1)

Input: grayscale eye image
Output: grayscale ROI image

procedure MIRT: LOCATE ROI
    $binary \leftarrow$ THRESHOLD($input$)
    $edges \leftarrow$ CANNY($binary$)
    DILATE($edges$)
    $largest, secondlargest \leftarrow$ FINDCONTOURS($edges$)
    if AREA(largest) < $AREA\_THRES$ **then** ▷ no valid contours found
        terminate
    else if AREA($secondlargest$) < $AREA\_THRES$ **then** ▷ only 1 contour found
        $c \leftarrow previous\_center.x$
        $r \leftarrow (c - x_{start})/(x_{end} - c)$
        if $largest.width > 2 * IRIS\_SIZE$ **then**
            $ROI \leftarrow$ RECTANGLE($largest$)
        else if $r < 1$ **then** ▷ right part visible
            $ROI.x_{start} \leftarrow largest.x_{start} - IRIS\_SIZE$
            $ROI.x_{end} \leftarrow largest.x_{start} + IRIS\_SIZE/2$
        else ▷ left part visible
            $ROI.x_{start} \leftarrow largest.x_{end} - IRIS\_SIZE/2$
            $ROI.x_{start} \leftarrow largest.x_{end} + IRIS\_SIZE$
    else ▷ 2 contour found
        $ROI \leftarrow$BOUNDINGRECTANGLE($largest, secondlargest$)
    CROP($input, ROI$) ▷ crop the input image to the ROI
    return ROI

the error between the estimated model and all feature points and count inliers/outliers. Algorithms 5 and 6 summarizes stage 2 procedure and how we use RANSAC in our system respectively. Figure 4.6 shows the steps of stage 2.



(a) Grayscale ROI

(b) Binary

(c) Closing/Opening

(d) Largest Contour / Feature Points

(e) Estimated Ellipse

Figure 4.6: Visulization of Mirt stage 2. (a) shows the input cropped ROI, (b) shows the binary image after inverse thresholding, (c) shows the binary image after performing closing followed by opening, (d) visualizes the largest contour in the image (colored with yellow) and extracted feature points marked with green crosshairs and red points, (e) shows the esmiated ellipse with red curve.

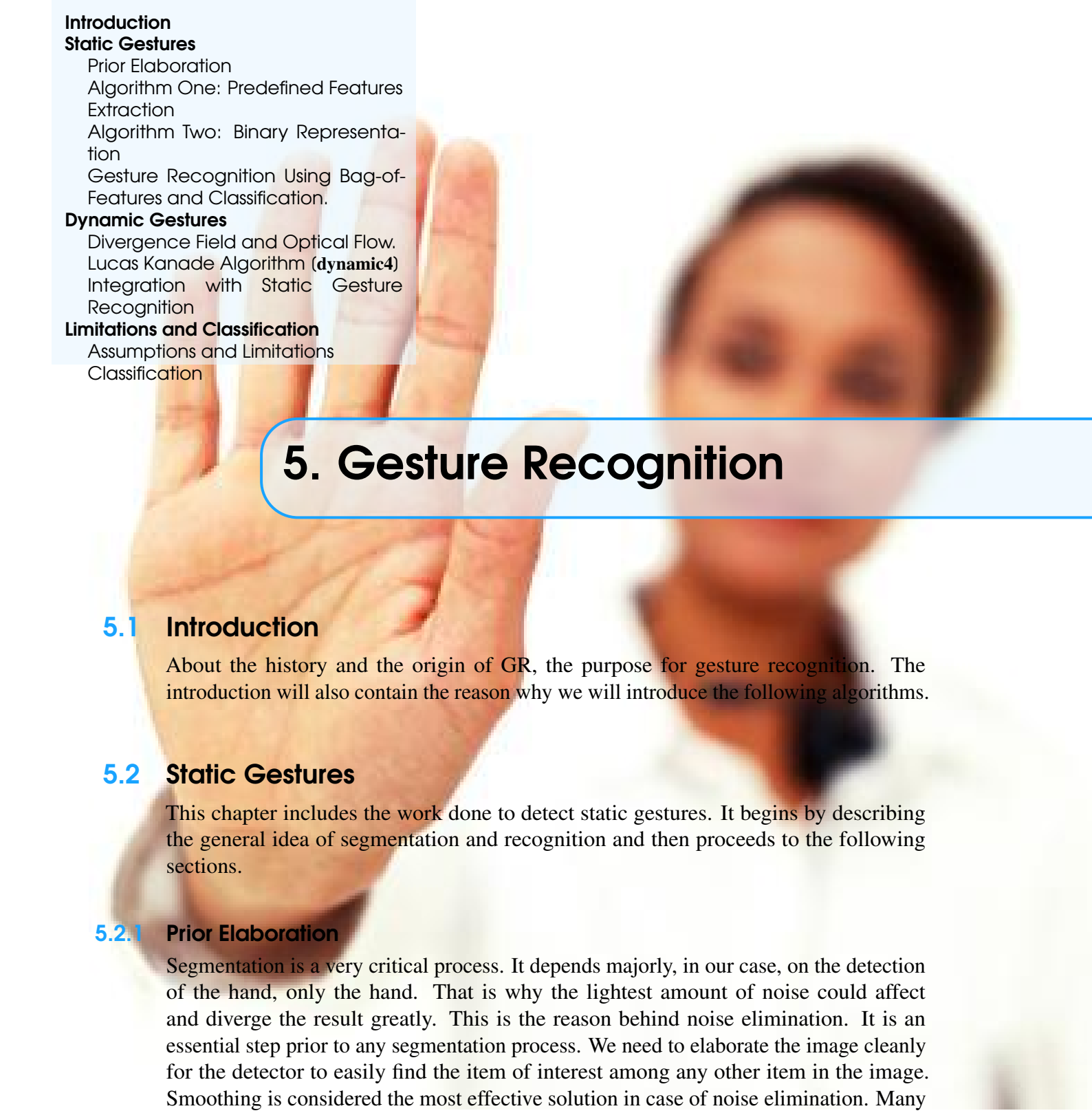
Algorithm 5 MIRT: Locate Iris (Stage 2)

Input: grayscale ROI image
Output: Ellipse that describes the iris

1: **procedure** MIRT: LOCATE IRIS
2: $binary \leftarrow$ INVERSETHRESHOLD($input$)
3: CLOSING($binary$)
4: OPENING($binary$)
5: $largestcontour \leftarrow$ FINDCONTOURS($binary$)
6: $ellipse \leftarrow$ RANSAC_ELLIPSE_FITTING($largestcontour$)
7: **return** $ellipse$

Algorithm 6 Our RANSAC Ellipse Fitting Procedure

```
1: procedure RANSAC_ELLIPSE_FITTING
2:     while i < N do
3:         Draw s points uniformly at random
4:         Fit ellipse model to these s points
5:         Find inliers to this ellipse (points whose algebraic error < t)
6:         If there are d or more inliers, accept the ellipse and refit using all inliers
```



5. Gesture Recognition

5.1 Introduction

About the history and the origin of GR, the purpose for gesture recognition. The introduction will also contain the reason why we will introduce the following algorithms.

5.2 Static Gestures

This chapter includes the work done to detect static gestures. It begins by describing the general idea of segmentation and recognition and then proceeds to the following sections.

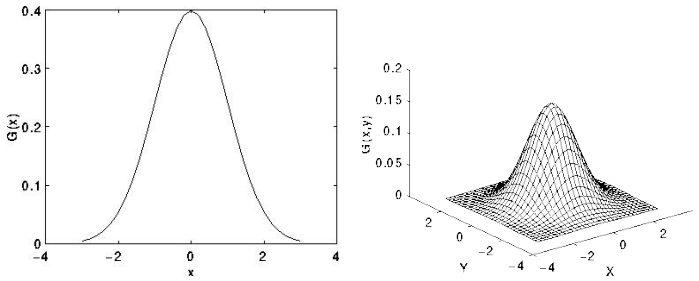
5.2.1 Prior Elaboration

Segmentation is a very critical process. It depends majorly, in our case, on the detection of the hand, only the hand. That is why the lightest amount of noise could affect and diverge the result greatly. This is the reason behind noise elimination. It is an essential step prior to any segmentation process. We need to elaborate the image cleanly for the detector to easily find the item of interest among any other item in the image. Smoothing is considered the most effective solution in case of noise elimination. Many approaches have been devised to perform smoothing capably on the images just to keep the meaningful item for us unharmed in case there is any interference of any noise. The mask, filter, we use for noise elimination is Gaussian Smoothing. need.

Gaussian Smoothing

The Gaussian smoothing operator is a 2-D convolution operator that is used to 'blur' images and remove detail and noise. In this sense, it is similar to the mean filter, but it uses a different kernel that represents the shape of a Gaussian ('bell-shaped') hump. This kernel has some special properties which are detailed below. The Gaussian distribution

in 1-D has the form:



(a) Gaussian Distribution (b) Gaussian Distribution in 2D

Figure 5.1: Gaussian Distributions

$$G(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{x^2}{2\sigma^2}} \tag{5.1}$$

where σ is the standard deviation of the distribution. We have also assumed that the distribution has a mean of zero (i.e. it is centered on the line x=0). The distribution is illustrated in Figure 5.1. In 2-D, an isotropic (i.e. circularly symmetric) Gaussian has the form:

$$G(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{5.2}$$

This distribution is shown in Figure 5.1 The idea of Gaussian smoothing is to use this 2-D distribution as a 'point-spread' function, and this is achieved by convolution. Since the image is stored as a collection of discrete pixels, we need to produce a discrete approximation to the Gaussian function before we can perform the convolution. In theory, the Gaussian distribution is non-zero everywhere, which would require an infinitely large convolution kernel, but in practice, it is effectively zero more than about three standard deviations from the mean, and so we can truncate the kernel at this point. Figure 3 shows a suitable integer-valued convolution kernel that approximates a Gaussian with a σ of 1.0. It is not obvious how to pick the values of the mask to approximate a Gaussian. One could use the value of the Gaussian at the center of a pixel in the mask, but this is not accurate because the value of the Gaussian varies non-linearly across the pixel. We integrated the value of the Gaussian over the whole pixel (by summing the Gaussian at 0.001 increments). The integrals are not integers: we rescaled the array so that the corners had the value 1. Finally, the 273 is the sum of all the values in the mask.

Once a suitable kernel has been calculated, then the Gaussian smoothing can be performed using standard convolution methods. The convolution can, in fact, be performed quickly since the equation for the 2-D isotropic Gaussian shown above is separable into x and y components. Thus, the 2-D convolution can be performed by first convolving with a 1-D Gaussian in the X direction, and then convolving with another 1-D Gaussian in the Y direction. (The Gaussian is in fact the only completely circularly symmetric operator that can be decomposed in such a way). Figure 5.2 shows the 1-D x component

kernel that would be used to produce the full kernel shown in Figure 5.2.1 (after scaling by 273, rounding and truncating one row of pixels around the boundary because they mostly have the value 0. This reduces the 7x7 matrix to the 5x5 shown above.). The Y component is exactly the same but is oriented vertically.

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

(a) The full kernel

.006	.061	.242	.383	.242	.061	.006

(b) 1-D x component kernel

Figure 5.2:

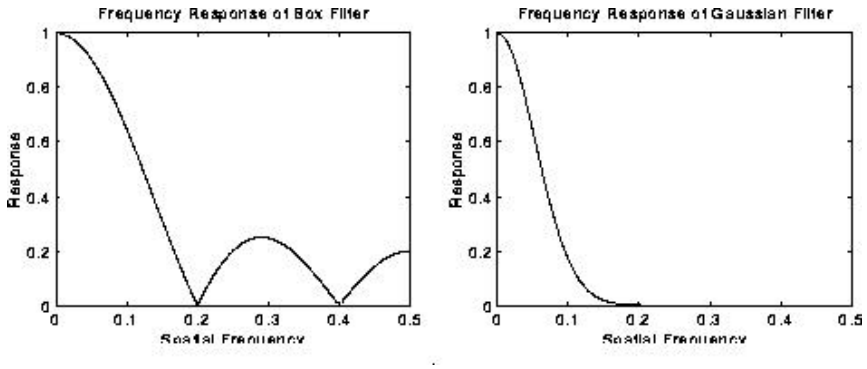
A further way to compute a Gaussian smoothing with a large standard deviation is to convolve an image several times with a smaller Gaussian. While this is computationally complex, it can have applicability if the processing is carried out using a hardware pipeline. The Gaussian filter not only has utility in engineering applications. It is also attracting attention from computational biologists because it has been attributed with some amount of biological plausibility, e.g. some cells in the visual pathways of the brain often have an approximately Gaussian response.

The effect of Gaussian smoothing is to blur an image, in a similar fashion to the mean filter. The degree of smoothing is determined by the standard deviation of the Gaussian. (Larger standard deviation Gaussians, of course, require larger convolution kernels in order to be accurately represented.)

The Gaussian outputs a 'weighted average' of each pixel's neighborhood, with the average weighted more towards the value of the central pixels. This is in contrast to the mean filter's uniformly weighted average. Because of this, a Gaussian provides gentler smoothing and preserves edges better than a similarly sized mean filter.

One of the principle justifications for using the Gaussian as a smoothing filter is due to its frequency response. Most convolution-based smoothing filters act as lowpass frequency filters. This means that their effect is to remove high spatial frequency components from an image. The frequency response of a convolution filter, i.e. its effect

on different spatial frequencies, can be seen by taking the Fourier transform of the filter. Figure 5.3 shows the frequency responses of a 1-D mean filter with width 5 and also of a Gaussian filter with $\sigma = 3$. We used figure 5.4 to illustrate the Gaussian's smoothing effect.



(a) Frequency responses

Figure 5.3: Frequency responses

(a) (b) (c)

Figure 5.4: Gaussian's smoothing effect

We used an appropriate value for σ and a kernel size 5*5 in our work. Of course, the higher kernel size we provide, the better smoothing we get, but the problem resides in the processing overhead. Since we have more processing coming head in several parts coming ahead we cannot just strain it and delay it here. Therefore, a secured limit of smoothing is sufficient to avoid the results that could harm the further work coming ahead.

Color Segmentation

Color image segmentation is useful in many applications. From the segmentation results, it is possible to identify regions of interest and objects in the scene, which is very beneficial to the subsequent image analysis or annotation. The problem of segmentation is difficult because of image texture. If an image contains only homogeneous color regions, clustering methods in color space are sufficient to handle the problem. In reality, natural scenes are rich in color and texture. It is difficult to identify image regions containing color-texture patterns.

Color images are usually represented in terms of their Red, Green and Blue components (RGB format) for purposes of storage and display. All of the color spaces can be derived from the RGB information supplied by devices such as cameras and scanners. The RGB color space is the most prevalent choice for computer graphics because color displays use red, green, and blue to create the desired color. Therefore, the choice of the RGB color space simplifies the architecture and design of the system. Furthermore, a system that is designed using the RGB color space can take advantage of a large number of existing software routines, since this color space has been around for a number of years.

However, RGB is not very efficient when dealing with "real-world" images. All three RGB components need to be of equal bandwidth to generate any color within the RGB color cube. The result of this is a frame buffer that has the same pixel depth and display resolution for each RGB component. In addition, processing an image in the RGB color space is usually not the most efficient method. For example, to modify the intensity or color of a given pixel, the three RGB values must be read from the frame buffer, the intensity or color calculated, the desired modifications performed, and the new RGB values calculated and written back to the frame buffer. If the system had access to an image stored directly in the intensity and color format, some processing steps would be faster. That is why we need desperately to move from RGB to another domain, where segmentation is more reliable. We have found it most beneficial to convert to YCBCr color space.

The YCbCr color space is widely used for digital video. In this format, luminance information is stored as a single component (Y), and chrominance information is stored as two color-difference components (Cb and Cr). Cb represents the difference between the blue component and a reference value. Cr represents the difference between the red component and a reference value.

YCbCr data can be double precision, but the color space is particularly well suited to uint8 data. For uint8 images, the data range for Y is [16, 235], and the range for Cb and Cr is [16, 240]. YCbCr leaves room at the top and bottom of the full uint8 range so that additional (nonimage) information can be included in a video stream. From YCbCr we apply a threshold on this color space to convert the image to the grayscale image.

The resulting conversion, based on a threshold, may result in a noise that we don't desire them to exist. This noise is totally hazard to the further processing of the image. Which means that we need another smoothing step. This time the smoothing requires

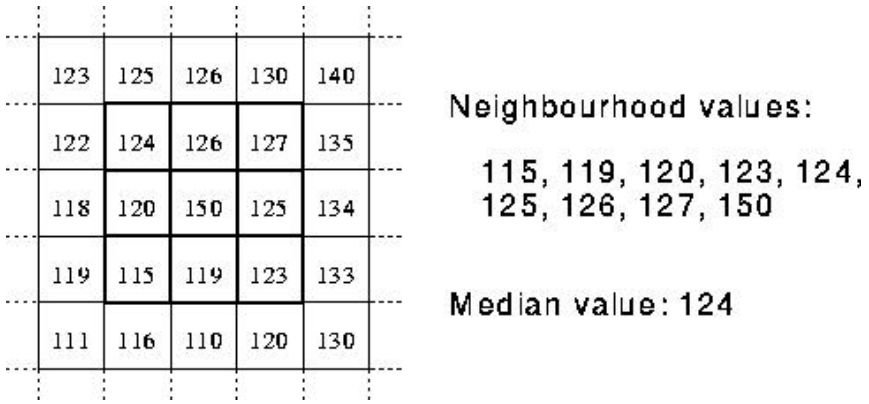
us to preserve certain details in the image, which are the edges of our items of interest. Consequently, we need a conservative smoothing that could secure us to preserve edges, which introduces us to the median filter.

Conservative Smoothing (Median Filter)

Conservative smoothing is a noise reduction technique that derives its name from the fact that it employs a simple, fast filtering algorithm that sacrifices noise suppression power in order to preserve the high spatial frequency detail (e.g. sharp edges) in an image. It is explicitly designed to remove noise spikes — i.e. isolated pixels of exceptionally low or high pixel intensity (e.g. salt and pepper noise) and is, therefore, less effective at removing additive noise (e.g. Gaussian noise) from an image.

Like most noise filters, conservative smoothing operates on the assumption that noise has a high spatial frequency and, therefore, can be attenuated by a local operation which makes each pixel's intensity roughly consistent with those of its nearest neighbors. However, whereas mean filtering accomplishes this by averaging local intensities and median filtering by a non-linear rank selection technique, conservative smoothing simply ensures that each pixel's intensity is bounded within the range of intensities defined by its neighbors.

The median filter is normally used to reduce noise in an image, somewhat like the mean filter. However, it often does a better job than the mean filter of preserving useful detail in the image. Like the mean filter, the median filter considers each pixel in the image in turn and looks at its nearby neighbors to decide whether or not it is representative of its surroundings. Instead of simply replacing the pixel value with the mean of neighboring pixel values, it replaces it with the median of those values. The median is calculated by first sorting all the pixel values from the surrounding neighborhood into numerical order and then replacing the pixel being considered with the middle pixel value. (If the neighborhood under consideration contains an even number of pixels, the average of the two middle pixel values is used.) Figure 5.5 illustrates an example calculation.



(a)

Figure 5.5:

By calculating the median value of a neighborhood rather than the mean filter, the median filter has two main advantages over the mean filter:

- The median is a more robust average than the mean and so a single very unrepresentative pixel in a neighborhood will not affect the median value significantly.
- Since the median value must actually be the value of one of the pixels in the neighborhood, the median filter does not create new unrealistic pixel values when the filter straddles an edge. For this reason, the median filter is much better at preserving sharp edges than the mean filter.

Figure 5.6 illustrates the effect of the median filter with many kernel sizes. At our work, we used the median filter with kernel size 5*5. It was most convenient for our work and this size preserved the edges the way we would like it to be.

(a) (b) (c)

Figure 5.6: Median Filter

Binary Image and Connected Components

NOT YET !!!

5.2.2 Algorithm One: Predefined Features Extraction

Feature extraction is a crucial step in computer vision applications for hand gesture recognition. The pre-processing stage prepares the input image and extracts features used later with the classification algorithms. This set of features shall be unique per ever gesture. The set of features the algorithm generates for training the dataset are as follow:

- Mean and Variance of the gray pixel values.
- The hand area and perimeter.
- Hand orientation.
- Orientation histogram.
- Radial signature.

(a) Mean and variance (b) Area and perimter

Figure 5.7: First Two Features

The first two features: mean, variance and the blob area and perimeter are detected from the previous steps prior to the algorithm. The smoothing followed by obtaining the gray image helps us finding the mean and the variance of the gray pixel values that represent the blob according to figure 5.7. The blob area and perimeter are detected from the last elaboration step which is finding the binary image for the blob. Figure 5.7 illustrates how this works. The area of the white region is calculated approximated and then the perimeter is determined on the position of the hand on the image. This now leaves us with the rest of the five other features remaining to find in the set.

Hand Orientation

The segmented hand is used to calculate its orientation with the help of image moments. Moment approximations provide one of the more obvious means of describing 2-D shapes. The moments involve sums over all pixels, and so are robust against small pixel changes. If I x, y is the image intensity at position x and y then the image moments, up to the second order, are:

$$M_{00} = \sum_x \sum_y I(x,y) \tag{5.3}$$

$$M_{01} = \sum_x \sum_y y.I(x,y) \tag{5.4}$$

$$M_{10} = \sum_x \sum_y x.I(x,y) \tag{5.5}$$

$$M_{11} = \sum_x \sum_y xy.I(x,y) \tag{5.6}$$

$$M_{20} = \sum_x \sum_y x^2 I(x,y) \tag{5.7}$$

$$M_{02} = \sum_x \sum_y y^2 I(x,y) \tag{5.8}$$

The hand position can be calculated as follows:

$$x_c = \frac{M_{10}}{M_{00}} \tag{5.9}$$

$$y_c = \frac{M_{01}}{M_{00}} \tag{5.10}$$

The hand orientation can then be calculated using the following intermediate variables a, b and c:

$$a = \frac{M_{20}}{M_{00}} - x_c^2 \tag{5.11}$$

$$b = 2(\frac{M_{10}}{M_{00}} - x_c y_c) \tag{5.12}$$

$$c = \frac{M_{02}}{M_{00}} - y_c^2 \tag{5.13}$$

and the angle is:

$$\theta = \frac{tan^{-1}(b,(a-c))}{2} \tag{5.14}$$

Orientation Histogram

Pixel intensities can be sensitive to lighting variations, which lead to classification problems within the same gesture under different light conditions. The use of local orientation measures avoids this kind of problem, and the histogram gives us translation invariance. Orientation histograms summarize how much of each shape is oriented in each possible direction, independent of the position of the hand inside the camera frame. This statistical technique is most appropriate for close-ups of the hand. In our work, the

hand is extracted and separated from the background. This provides a uniform black background, which makes this statistical technique a good method for the identification of different static hand poses. This method is insensitive to small changes in the size of the hand, but it is sensitive to changes in hand orientation.

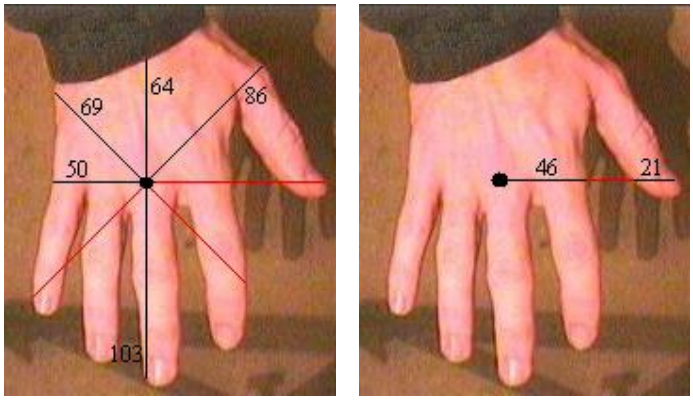
We have calculated the local orientation using image gradients, represented by horizontal and vertical image pixel differences. If dx and dy are the outputs of the derivative operators, then the gradient direction is $arctan(dx, dy)$ and the contrast is $\sqrt{d_x^2 + d_y^2}$ A contrast threshold is set as some amount k times the mean image contrast, below which we assume the orientation measurement is inaccurate. A value of k=1.2 was used in the experiments. We then blur the histogram in the angular domain as in with a [1 4 6 4 1] filter, which gives a gradual fall-off in the distance between orientation histograms.

Our histogram of orientation is composed of 36 bins, which means that this feature is described via 36 values.

Radial Signature

A simple method to assess the gesture would be to measure the distance from the hand centroid to the edges of the hand along a number of radials equally spaced around a circle. This would provide information on the general "shape" of the gesture that could be easily rotated to account for hand yaw (since any radial could be used as datum). Figure 5.8 (a) shows a gesture with example radials (simplified). However, a problem

(as shown in Figure 5.8 (a)) is how to measure when the radial crosses a gap between fingers or between the palm and a finger. To remedy this it was decided to count the total number of skin pixels along a given radial. This is shown in Figure 5.8 (b). All of the radial measurements could then be scaled so that the longest radial was of constant length. By doing this, any alteration in the hand camera distance would not affect the radial length signature generated.

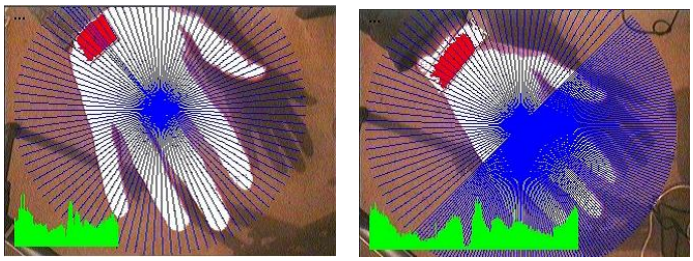


(a) Radial Idea (b) Corrected Radial

Figure 5.8: Radial Signature

A simple method to assess the gesture would be to measure the distance from the hand centroid to the edges of the hand along a number of equally spaced radials. For the present feature extraction problem, 100 equally spaced radials were used. To count the number of pixels along a given radial we only take into account the ones that are part of the hand, eliminating those that fall inside gaps, like the ones that appear between fingers or between the palm and a finger.

During tests it was noticed that the quality of recognition depended on the number of radials used. It was also noticed that most of the significant data was concentrated around the fingers, thus it would be more efficient to group radials in these areas. Figure 5.9 shows the radials in their original grouping and after reorganization.



(a) Radial Signature with bin=100 (b) Radial Signature with bin=100

Figure 5.9: Radial Signature with different bin sizes

All the radial measurements can be scaled so that the longest radial has a constant length. With this measure, we can have a radial length signature that is invariant to hand distance from the camera.

5.2.3 Algorithm Two: Binary Representation

The algorithm is based on describing the image, compare this description, and classify it with the help of the data set after walking through the previously mentioned steps for elaborating the image and extracting the hand in binary representation, an example for a binary image is shown in figure 5.10. Standardly, we now have an image that contains two meaningful details only the foreground pixels that should only represent the hand and the background pixels that only represent the background. Generally, to describe an image we need every pixel of image to declare itself, vote whether it is foreground or background, and then we can have the general visualization for the image by traversing ever pixel and checking what it looks like. Therefore, to traverse ever pixel this is a huge processing consumption given that we have many frames coming in it is needless to do a per pixel checking for each image in each frame.

Our algorithm alleviates such processing burden. The general idea of the algorithm includes this voting but in a less processing way. We need parts of the image to vote on their existence but not pixel by pixel. Given the binary image, it is quite visible that most of the pixels are going to declare themselves a background pixel or, at the best fitting, we find the same number of pixels on foreground and background. However,

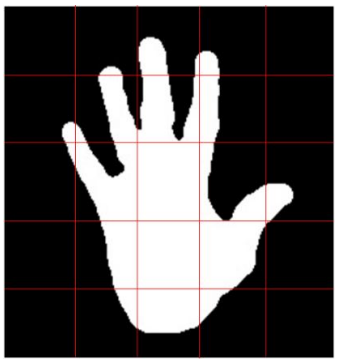
(a) Open Hand

Figure 5.10: Binary Image Example

this is still a waste of processing. The algorithm we are about to detail its idea starts by introducing what can resolve this needless processing waste, the griding.

Griding The Binary Image

Our Work begins by dividing the binary image we obtained from our elaboration steps into grids as shown in Figure 5.11. The griding is an alternative approach for pixel-by-pixel traversing. In this approximation, we just process n grids, standardly, rather than over thousands and thousands of pixels. This solution is based on the nature of the input image as a binary image. It may not work if the image is not binary image. The binary representation helps us limit any grid value to either 0 or 1 according to the voting technique we use.

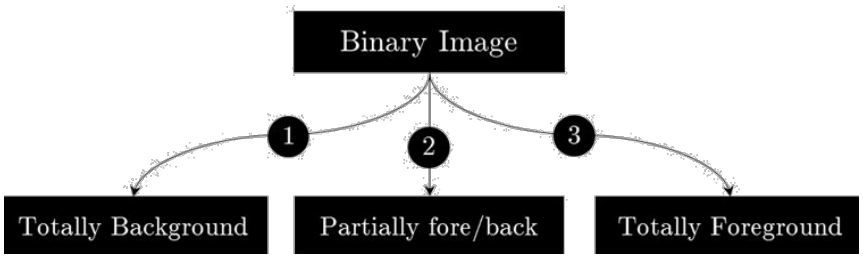


(a)

Figure 5.11: Griding Idea Example

Grid Voting

As the image is a binary image, some grids vote according to one of the following cases in Figure 5.12



(a)

Figure 5.12: Grids Vote Cases

1st Case: Totally Background

In this case, the whole grid is formed by background pixels only as shown in Figure 5.13 (a), all of them are black pixels. Consequently, the grid votes on its value to be 0. Note that to know this nature about a grid we traverse its pixels. This is not reverting us back to the pixel-by-pixel approach but, here, we just traverse until we encounter the existence of a one white pixel only. This white pixel is always interpreted as a foreground pixel and this moves us to the second case. Therefore, we are not traversing all the pixels this just happens in the grids that contain no foreground pixels but generally, the voting is determined after the traversing is done once we hit the end of the grid and no foreground pixels exist or once we come across a foreground pixel. Which guarantees the running time and the processing time to be kept lower than the former approach that counts the voting of ever pixel.

(a) Case 1 (b) Case 2 (c) Case 3

Figure 5.13: Grid Shapes

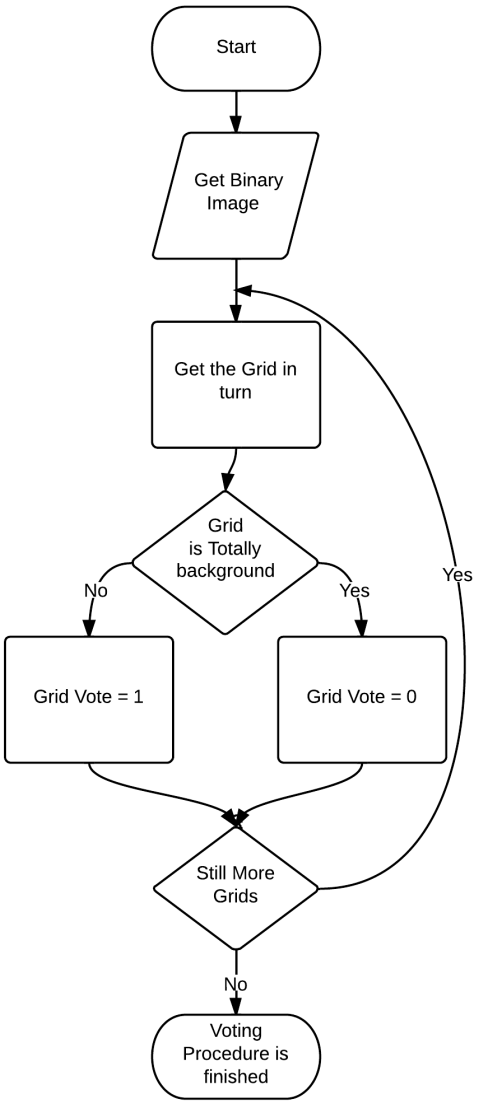
2nd Case: Partially Background

Different from the first case, the image's grid, in this one, encounters the presence of two kinds of pixels; background and foreground. Check examples in Figure 5.13. In this case, the voting is determined directly, without traversing, if the first pixel in the grid is a foreground pixel. However there could be a background pixels in the upcoming parts of the grid but we don't need to traverse for the rest of pixels, as the voting immediately turns to "1" once we find this white pixel. Otherwise, if the first pixel is background pixel, hence, we begin in the same way as the first case until we hit the presence of a foreground pixel then the traversing is terminated and the vote of the whole grid turns to "1". If no foreground pixels detected, then the voting remains "0" as in the first case. The grid that can be seen as an edge grid is the one that can be described via this case.

The traversing stops at the yellow points which explains the variety of this case. All these sub-cases are shown in Figure 5.13 (b).

3^{rd} Case: Totally Foreground

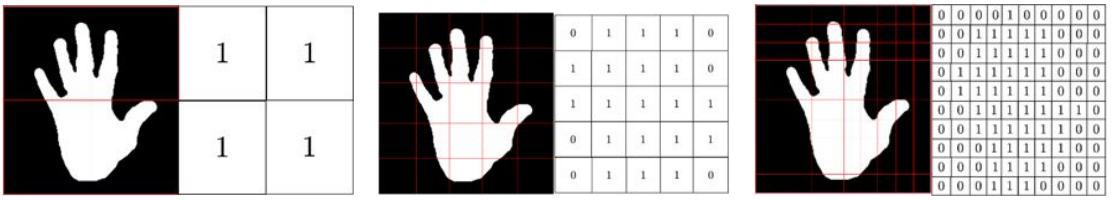
There is no much difference between this case and the previous one. In both cases, we encounter the presence of foreground pixels but this case can be considered a special case from the previous case since we have no background pixels. Which means that the grid is totally foreground. This can be interpreted that the grid is not an edge grid, as it has nothing from the background. As show in Figure 5.13(c). The Whole case and the vote determination is explained by the flow chart in Figure 5.14.



(a)

Figure 5.14: Voting Flow Chart

The griding approximation is acting as if we let group of pixels decides rather than pixel by pixel. On the one hand, this counts best for the processing performance, much better, but on the other hand, that doesn't sound promising for the accuracy. To illustrate this clearly let us consider and talk about the following example if we choose the griding to be only 2*2 grids. At this case, we may have the binary shape of the hand, in any form. While detection, it will always results in a just four grids and all of them are voted to be 1. Another case, if we used 5*5 grids. This results in a completely different voting result. Furthermore, if we used 10*10 grids, the resulting voting is presented in the Figure 5.15 (a,b,c)



(a) n=2 (b) n=5 (c) n=10

Figure 5.15: Voting Results Using Different Grid Sizes

The pixel-by-pixel detection gives us an optimal description and yet the worst performance but we can approximate our approach to go nearer to this optimal description and yet preserves the performance. This can be achieved if we choose a larger number of grids that can actually be considered as if they are pixels not girds. The larger we keep the number of girds growing to be, the better accuracy we obtain, and the worse the performance gets. This keeps happening as long as we increase the number of girds until the grid is actually a pixel, then, the worst performance occur and the best accuracy is obtained.

The reason why this happens can be see clearly in table (Not Yet). When we increase the grid size, we happen to include more details in each grid that can be described optimally and totally by pixel-by-pixel approach. As long as we make the grids come across edges in a way that griding attempts to preserve the shape of the binary image, our accuracy will rapidly grow. Therefore, to conclude, more girds better accuracy. Every time we enlarge the gird size we check the accuracy and performance until we hit this number of grids that gives acceptable results on each as recorded in table (Not Yet).

The previous figures also assert what we were saying about the indispensability of noise elimination. Since, if we got one white noisy spot in the image it will significantly affect the resulting matrix and affect all that is following this step.

Flatting The Votes

After each gird's voting is determined, now we get a matrix of 0s and 1s. This matrix will be formulated as 1-D array that Represent the feature array which will be passed to the classifier either for training/evaluating.

We get to detail how the classifiers we used to evaluate and test our algorithm work in the classification chapter.

5.2.4 Gesture Recognition Using Bag-of-Features and Classification.

Introduction

Content based image retrieval (CBIR) is still an active research field. There are a number of approaches available to retrieve visual data from large databases. But almost all the approaches require an image digestion in their initial steps. Image digestion is describing an image using low level features such as color, shape, and texture while removing unimportant details. Color histograms, color moments, dominant color, scalable color, shape contour, shape region, homogeneous texture, texture browsing, and edge histogram are some of the popular descriptors that are used in CBIR applications. Bag-Of-Feature (BoF) is another kind of visual feature descriptor which can be used in CBIR applications. In order to obtain a BoF descriptor we need to extract a feature from the image. This feature can be any thing such as SIFT (Scale Invariant Feature Transform), SURF (Speeded Up Robust Features), and LBP (Local Binary Patterns), etc.

In this system we are using the Scale Invariant Feature Transform (SIFT) features [**sift**]. These visual features/keypoints allow for reliable matching between different views of the same object, image classification, and object recognition. The SIFT features/keypoints are invariant to scale, orientation, and partially invariant to illumination changes, and are highly distinctive of the image. However, SIFT features are too high dimensionality to be used efficiently. We propose to solve this problem by using the bag-of-features approach [**Lazebn06**] to reduce the dimensionality of the feature space.

Features Extraction Using SIFT

Features based on the SIFT algorithm are invariant to scale and rotation and can be extracted in real-time for low resolution images. They are extracted in four stages. The first step finds the locations of potential interest points in the image by detecting the maxima and minima of a set of difference of Gaussian filters applied at different scales all over the image. Then, these locations are refined by eliminating points of low contrast. An orientation is then assigned to each keypoint based on local image features. Finally, a local feature descriptor is computed at each keypoint. This descriptor is based on the local image gradient transformed according to the orientation of the keypoint for providing orientation invariance. The size of the feature vector depends on the number of histograms and the number of bins in each histogram. In Lowe's original implementation[**sift**] a 4-by-4 patch of histograms with 8 bins each is used, generating a 128-dimensional feature vector. See Figure 5.16 for example of extracted features. Each features is descripted using 1D array of numbers.

K-Means Clustering

Clustering divides a group into subgroups, called clusters, so that the elements in the same cluster are similar. It is implemented using an unsupervised learning algorithm and an ordinary method for statistical data analysis applied in several fields, such as machine learning, pattern recognition, image analysis, data mining, and bioinformatics.

The number of clusters (the codebook size) depends on the structure of the data. There will be a compromise for how to choose the vocabulary size or number of clusters. If it is too small, then each bag-of-words vector will not represent all of the keypoints

(a) Open Hand (b) Closed Hand (c) Hand pointing to left

Figure 5.16: SIFT features

extracted from its related image. If it is too large, it will lead to overfitting because of insufficient samples of the keypoints extracted from the training image. We trained the classifier using 234 images with number of bags equal to 200. This number provides the most accurate recognition rate by trying different values [**Dardas11**].

The first step in k-means clustering is to divide the vector space (128-dimensional feature vector) into k clusters. K-means clustering starts with k randomly located centroids (points in space that represent the center of the cluster) and assigns every keypoint to the nearest one. After the assignment, the centroids (codevectors) are shifted to the average location of all the keypoints assigned to them, and assignments are redone. This procedure repeats until the assignments stop changing.

Once this is done, each feature vector (keypoint) is assigned to one and only one cluster center that is in the nearest distance with respect to the Euclidean distance metric in the 128- dimensional feature vectors. The keypoints that are assigned to the same cluster center will be in the same subgroup so that after clustering, we have k disjoint subgroups of keypoints. Therefore, k-means clustering decreases the dimensionality for every training image with n keypoints (nx128) to 1xk, where k is the number of clusters.

Classification

All the keypoints recovered in the training images are mapped with its generated bag-of-words vector using k-means clustering. Then, all bag-of-words vectors with their related class or label numbers are fed into classifier. The following Algorithm 7 sums up the whole process. After the classifier is trained. For each new frame, SIFT features are

extracted then clustered. Then the classifier is used to classifiy the clustered features. See the following Algorithm, Algorithm 8

Different classifiers were used and results were recorded. See chapter Limitations and classifications for more details.

Algorithm 7 Bag-of-Features SIFT Clustering

Input: Dataset
Output: Set of fixed attributes for each image
procedure BAG-OF-FEATURES SIFT CLUSTERING

Stage 1: Obtain the set of bags of features.
for all images in the dataset **do**
Extract the SIFT feature points.
Obtain the SIFT descriptor the feature points extracted from the image.

Stage 2: Clustering
Set the amount of bags to chosen number.
Cluster the set of feature descriptors.
Train the bags with K-Means Algorithm.
Obtain the visual vocabulary.

Algorithm 8 Classification of given image

Input: Image
Output: Gesture label if found
1: **procedure** CLASSIFICATION OF GIVEN IMAGE

2: Extract SIFT feature points of the given image.
3: Obtain SIFT descriptor for each feature point.
4: Match the feature descriptors with the vocabulary we created in the first step
5: Classifiy.

5.3 Dynamic Gestures

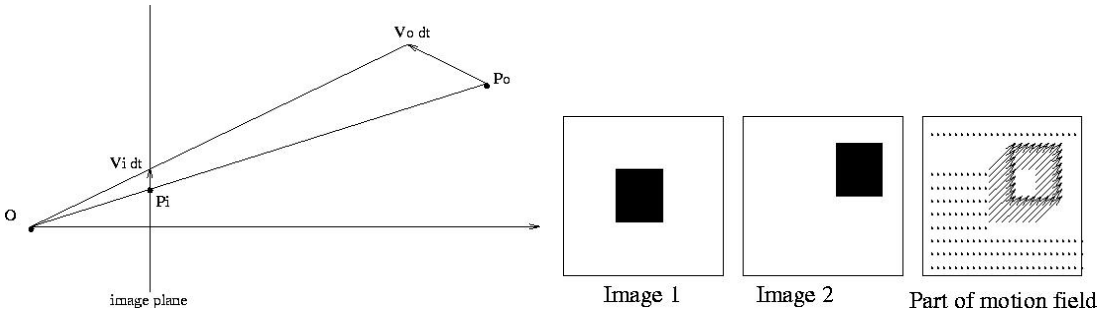
A lot of information can be extracted from time varying sequences of images, often more easily than from static images. In our case, a huge extension and variety of applications can be based on the detection of a moving hand. Camouflaged objects, the hand, are only easily seen when they move. Moreover, the relative sizes and position of objects are more easily determined when the objects move. Even simple image differencing provides an edge detector for the silhouettes of texture-free objects moving over any static background.

The analysis of visual motion divides into two stages:
- The measurement of the motion, and
- The use of motion data to segment the scene into distinct objects and to extract two-dimensional information about the shape and motion of the objects [**dynamic1**].

There are two types of motion to consider; movement in the scene with a static camera, and movement of the camera, or ego motion. Since motion is relative anyway, these types of motion should be the same. However, this is not always the case, since if the scene moves relative to the illumination, shadow effects need to be dealt with. In addition, specularities can cause relative motion within the scene. For our work, we

will ignore all such complications and focus on the first case which is having our hand moving before a static camera.

When an object moves in front of a camera, there is a corresponding change in the image. Thus, if a point p_0 on an object moves with a velocity v_0, then the imaged point p_i can be assigned a vector v_i to indicate its movement on the image plane as Showen in Figure 5.17(a). The collection of all these vectors forms the motion field.



(a) Point Moving in Image Plane (b) Motion Field Idea

Figure 5.17:

If we are only dealing with rigid body translations and rotations, then the motion field will be continuous except at the silhouette boundaries of objects.

In the case of pure camera translation, the direction of motion as in Figure 5.17 (b) is along the projection ray through that image point from which (or towards which) all motion vectors radiate. The point of divergence (or convergence) of all motion field vectors is called the Focus Of Expansion FOE (or focus of contraction FOC). Thus, in the case of divergence we have forward motion of the camera, and in the case of convergence, backwards motion.

If we take the axis of camera translation as the camera baseline in stereo, then every projection of a fixed scene point must translate along an epipolar line, and all such lines converge at the epipole, which is just the FOE [**dynamic2**]. Therefore, in order to track the motion we firstly need to determine the divergence field of optical flow motion.

5.3.1 Divergence Field and Optical Flow.

In a vector field, divergence is an operator that measures the magnitude of the source or sink of the field. Given a vector $F = [F_1, F_2, .., F_n]^T$ in a n-dimensional Euclidean space, the divergence of F can be calculated as

$$DIVF = \sum_{i=1}^{n} \frac{dF_i}{dx_i} \tag{5.15}$$

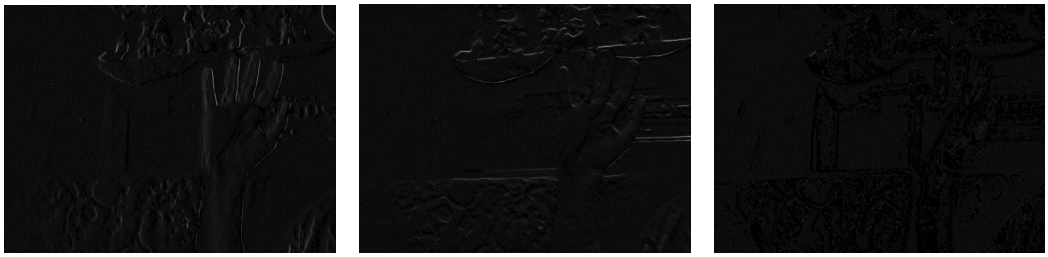
Where $[x1, x2, x3 \ldots xn]$ T are the Cartesian coordinates of the space where the vector field is defined.

Accordingly, for an optical flow vector field $F(x,y) = [u(x,y), v(x,y)]T$ where $u(x,y)$ and $v(x,y)$ are respectively the horizontal and vertical components of optical flow at position (x,y) the divergence of F is:

$$divF = \frac{du}{dx} + \frac{dv}{dy} \tag{5.16}$$

The following figures present an example of transforming s into a divergence field, where:

- Figure 5.18 (a,b) are Visualization of u and v respectively.
- Figure 5.18 (c) The corresponding Divergence Field after being normalized to gray-scale [**dynamic3**].



(a) (b) (c)

Figure 5.18: Example of transforming S into A divergence Field

Then we use this as an input for Lucas Kanade Algorithm to calculate the Optical flow. In computer vision terms, Optical Flow is the image motion of objects as the objects, scene or camera, moves between two consecutive images. It is a 2D vector field of within-image translation. It is a classic and well-studied field in computer vision with many successful applications in for example video compression, motion estimation, object tracking and image segmentation.

Optical flow relies on three major assumptions:

- Brightness constancy: The pixel intensities of an object in an image does not change between consecutive images.
- Temporal regularity: The between-frame time is short enough to consider the motion change between images using differentials (used to derive the central equation below).
- Spatial consistency: Neighboring pixels have similar motion.

In many cases these assumptions break down, but for small motions and short time steps between images it is a good model. Assuming that an object pixel I(x,y,t) at time t has the same intensity at time $t + \delta_t$ after motion $[\delta_x, \delta_y]$ means that

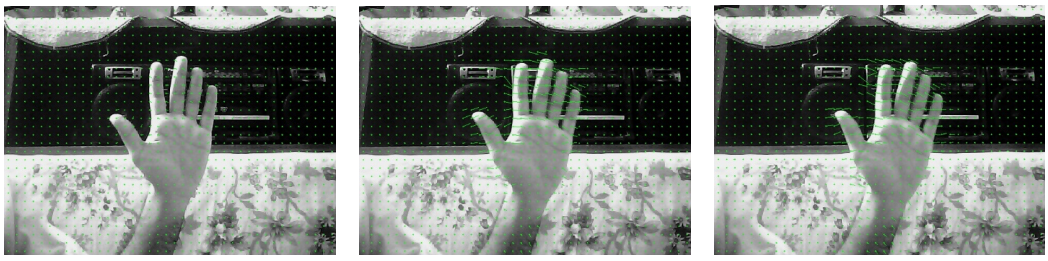
$$I(x,y,t) = I(x+\delta_{x,y}+\delta_{y,t}+\delta_t). \tag{5.17}$$

Differentiating this constraint gives the optical flow equation:

$$\bigtriangledown I^T V = -I_T. \tag{5.18}$$

Where $V = [u,v]$ is the motion vector and It the time derivative. For individual points in the image, this equation is under-determined and cannot be solved (one equation with two unknowns in v). By enforcing some spatial consistency, it is possible to obtain solutions though. In the Lucas-Kanade algorithm below we will see how that assumption is used.

The following figures show optical flow vectors (sampled at every 16^th pixel) shown on video of a waving hand [**dynamic4**].



(a) (b) (c)

Figure 5.19:

(a) (b) (c)

Figure 5.20:

5.3.2 Lucas Kanade Algorithm (dynamic4)

Tracking is the process of following objects through a sequence of images or video. The most basic form of tracking is to follow interest points such as corners. A popular algorithm for this is the Lucas-Kanade tracking algorithm which uses a sparse optical flow algorithm. Lucas-Kanade tracking can be applied to any type of features but usually makes use of corner points similar to the Harris corner points [**dynamic6**] .

Depending on Harris corner points we find good feature points to track. These points are corners detected according to an algorithm by Shi and Tomasi [**dynamic7**] where corners are points with two large eigenvalues of the structure tensor (Harris matrix) and where the smaller eigenvalue is above a threshold.

The optical flow equation is under-determined (meaning that there are too many unknowns per equation) if considered on a per-pixel basis. Using the assumption that neighboring pixels have the same motion it is possible to stack many of these equations into one system of equations

For some neighborhood of n pixels. This has the advantage that the system now has more equations than unknowns and can be solved with least square methods. Typically, the contribution from the surrounding pixels is weighted so that pixels farther away have less influence. A Gaussian weighting is the most common choice.

Standard Lucas-Kanade tracking works for small displacements. To handle larger displacements a hierarchical approach is used. In this case, the optical flow is computed at coarse to fine versions of the image. The following Figures shows the steps of Lucas-Kanade Tracking Algorithm. First by detecting the feature points in the first image and then keep tracking the flow until the motion is 0 then draws where the feature points have stopped.

(a) (b) (c)

Figure 5.21:

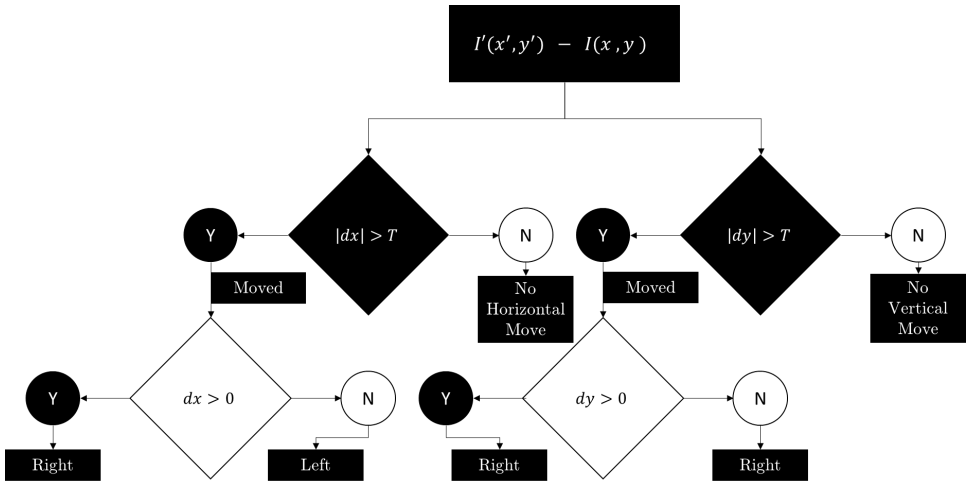
From this stage, our algorithm exploits the Lucas-Kanade Algorithm and takes a very simple turn. Based on the assumption mentioned at the beginning of the chapter, we only have four directions for the motion; Up, Down, Right Left. The simplicity of our dynamic gestures alleviated the processing overhead gigantically and yet made great use of the output of Lucas-Kanade Algorithm.

From Lucas-Kanade Algorithm we get an output as 2D array. This 2D array contains the new positions for the detected feature point as tracked. Given that we already have the original indices of these feature points. We can get the difference between the coordinates of the new positions and the old ones.

From Lucas-Kanade Algorithm we get an output as 2D array. This 2D array contains the new positions for the detected feature point as tracked. Given that we already have the original indices of these feature points. We can get the difference between the coordinates of the new positions and the old ones.

To decide that the hand has moved, the difference between the new position and the new one must exceed a given threshold. If the hand even slightly moved and the difference is less than the threshold it is considered static until the movement exceeds the threshold then the detection starts. This threshold is applied to the vertical and horizontal motion components for ever feature point. If the x or the horizontal motion difference exceeded the threshold then the hand object moved horizontally. The same applies to y or the vertical component. If both differences at the x and y components didn't exceed the threshold, then this feature points hadn't move.

The following flowchart explains what happens at each feature point, where $I(x,y)$ is the original indices for a feature point and $I'(x',y')$ the new indices in the new image after the movement.



(a)

Figure 5.22: Flow Chart

Then we check the difference itself if it is indicating that the hand moved if the direction is to the positive side of each axes or differently in the negative one. We have four counters to help in determining the direction. Each counter is increased at each feature point in the direction this point moved to.

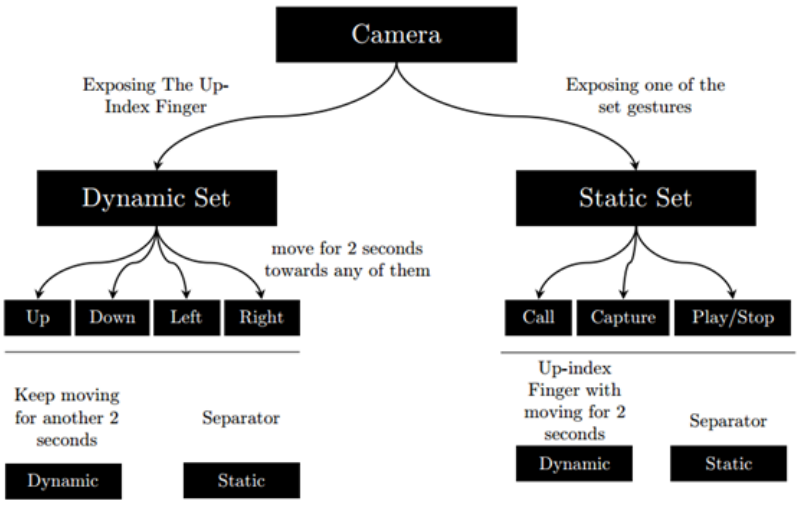
According to these two steps of the flowchart at each feature point one or two of the four counters are increased. This includes if a point moved in two directions i.e. bottom left. The point that according to the movement to the right, left up or down, the majority of points will vote correctly and the counter of the correct direction will always be greater than the rest.

5.3.3 Integration with Static Gesture Recognition

The Gesture Recognition Process always starts with simple static gesture recognition. The recognition of the static gestures goes smoothly until the up-index finger appear before the camera. This signals the detection of a dynamic gesture coming next. The algorithm then wait for another frame with the same gesture but with its feature points moved to another location. The algorithm keeps taking frames of the dynamic gestures, applying optical flow algorithm for every two consecutive frames. Consequently, it stores the direction until there is no motion between the upcoming frames (up to three frames with no motion).Then, take the action according to the majority of the saved directions.

The handling of static gestures can be reactivated once more, if the separator appeared which indicates a change of gestures. The dynamic gestures has no separator as long as you keep going in front of the camera your motions gets recorded and interpreted according to the application.

The following diagram illustrates these transitions:



(a)

Figure 5.23: Transition Diagram Between Static and Dynamic Gestures

5.4 Limitations and Classification

This chapter includes the work done to detect static gestures. It begins by describing the general idea of segmentation and recognition and then proceeds to the following sections.

5.4.1 Assumptions and Limitations

Not Yet

5.4.2 Classification

Our Dataset of gestures contains exactly 234 images. These images are different views for our 8 gestures. Every view is represented as a 1-D Array. The group of images or views is acting as a reference for the classifier to check every query image against it. For every gesture there is a label that the classifier, when its job is done, decides, according to this classifier's algorithm, to which label the query image belongs. The following is our gestures and with how many views they are represented in the dataset:

Start Gesture

The gesture is represented by the Ok sign (3 fingers) as shown in Figure 5.24 (a). This gesture is marking the start of the detection for any gesture to come forth. It is represented with 18 views in the dataset.

Open Hand

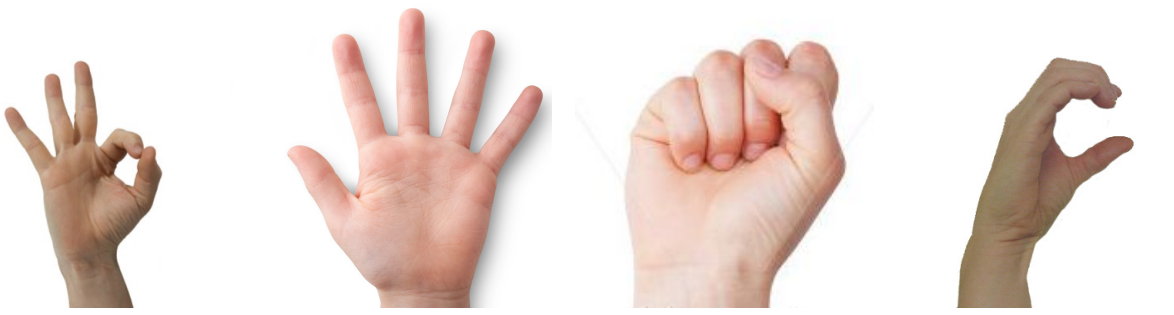
This gesture is represented by the full hand opened and exposed to the camera as show in Figure 5.24 (b). This gesture can be interpreted in many ways according to the application. For example, stop and play in music player. In the dataset, this gesture is represented by 22 views including left and right hands.

Closed Hand

This gesture is represented by the full hand closed, fist, and exposed to the camera as show in Figure 5.24(c). This gesture has a special rule. It acts as a separator between the gestures detection i.e. if we detected a gesture, we cannot hope to detect one another following it until the algorithm, any of them receives the fist gesture. In the dataset, this gesture is represented by 14 views.

Capture Gesture

This gesture is represented by a hand shaping a letter C as show in Figure 5.24(d). This gesture has a specific need which is to signal the camera to be opened for capturing an image and capturing after the camera is running. In the dataset, this gesture is represented by 54 view including left and right hands.



(a) Start Gesture (b) Open Hand Gesture (c) Closed Hand Gesture (d) Capture Gesture

Figure 5.24: Defined Gestures Set 1

Call Gesture

This gesture is represented by the thumb and the little finger and exposed to the camera as show in Figure 5.25(a). This gesture is used specifically to open the dialer or actually signal to call a number. In the dataset, this gesture is represented by 31 view including left and right hands.

Up Gesture

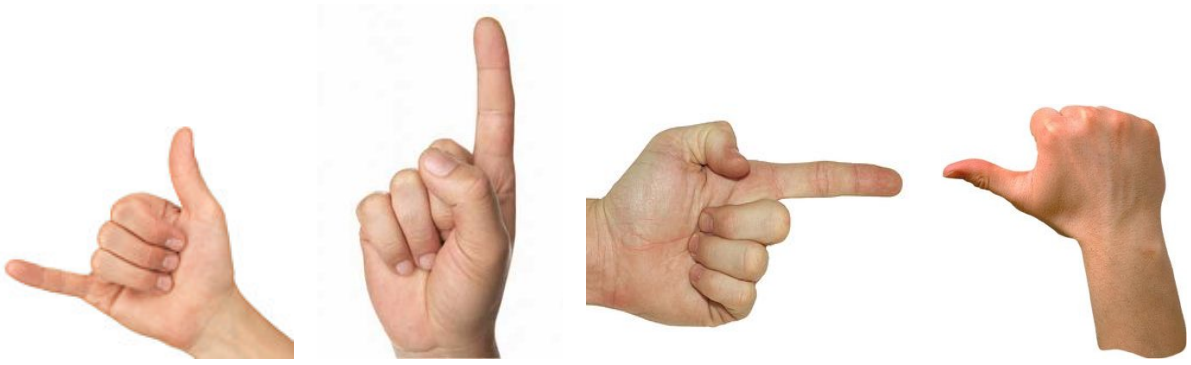
This gesture is represented index finger pointing up and exposed to the camera as show in Figure 5.25(b). This gesture has a special role in our set. It marks the start of detecting a dynamic gesture as mentioned in the dynamic gesture chapter. In the dataset, this gesture is represented by 24 view including left and right hands.

Right Gesture

This gesture is represented by the thumb pointing to the right and exposed to the camera as show in Figure 5.25(c). This gesture can be interpreted in many ways according to the application. For example, stop and play in music player. In the dataset, this gesture is represented by 30 views.

Left Gesture

This gesture is represented by the thumb pointing to the left and exposed to the camera as show in Figure 5.25(d). This gesture can be interpreted in many ways according to the application. For example, stop and play in music player. In the dataset, this gesture is represented by 22 views.



(a) Call Gesture (b) Up Gesture (c) Right Gesture (d) Left Gesture

Figure 5.25: Defined Gestures Set 2

Every algorithm from the previous chapter passes a 1-D array to the classifier. This 1-D array is the classifier input. In the following sections, we are going to represent the three classifiers we used, for our work and how they use this 1-D array.

Support Vector Machine

SVM method for the classification of both linear and nonlinear data [**classifications**]. The algorithm works as follows. It uses a nonlinear mapping to transform the original training data into a higher dimension. Within this new dimension, it searches for the linear optimal separating hyperplane (i.e., a “decision boundary” separating the tuples

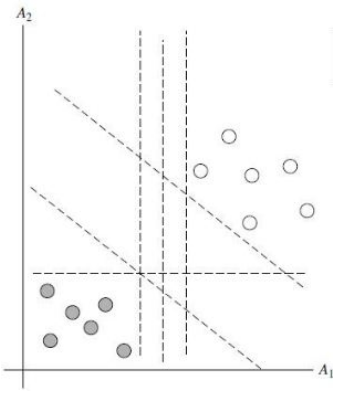
of one class from another). With an appropriate nonlinear mapping to a sufficiently high dimension, data from two classes can always be separated by a hyperplane. The SVM finds this hyperplane using support vectors ("essential" training tuples) and margins (defined by the support vectors).We will delve more into these new concepts later.

Although the training time of even the fastest SVMs can be extremely slow, they are highly accurate, owing to their ability to model complex nonlinear decision boundaries. They are much less prone to overfitting than other methods. The support vectors found also provide a compact description of the learned model. SVMs can be used for numeric prediction as well as classification. They have been applied to a number of areas, including handwritten digit recognition, object recognition, and speaker identification, as well as benchmark time-series prediction tests.

Linear Separability Of The Data

To explain the mystery of SVMs, let's first look at the simplest case; a two-class problem where the classes are linearly separable. Let the data set D be given as (X1, y1), (X2, y2), ... , (XD, yD), where Xi is the set of training tuples with associated class labels, yi. Each yi can take one of two values, either +1 or -1 (i.e., $yi \subseteq +1, -1$), corresponding to the classes buys computer D yes and buys computer D no, respectively.

To aid in visualization, let's consider an example based on two input attributes, A1 and A2, as shown in Figure 5.26. From the graph, we see that the 2-D data are linearly separable (or "linear," for short), because a straight line can be drawn to separate all the tuples of class +1 from all the tuples of class -1.

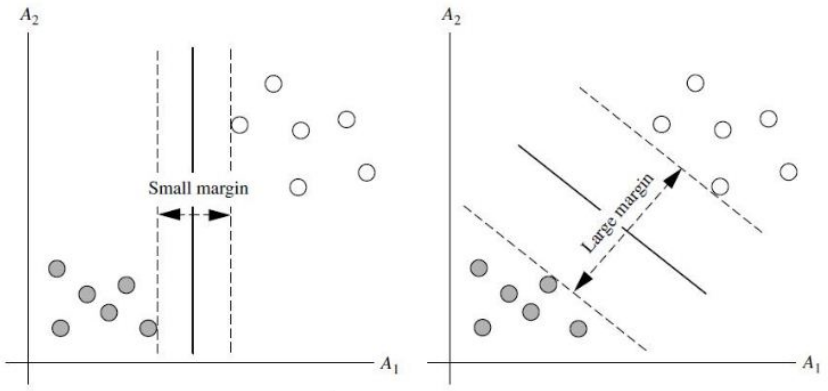


(a)

Figure 5.26: SVM Separability Idea

There are an infinite number of separating lines that could be drawn. We want to find the "best" one, that is, one that will have the minimum classification error on previously unseen tuples. How can we find this best line? Note that if our data were 3-D (i.e., with three attributes); we would want to find the best separating plane. Generalizing to n dimensions, we want to find the best hyperplane. We will use "hyperplane" to refer to the decision boundary that we are seeking, regardless of the number of input attributes.

An SVM approaches this problem by searching for the maximum marginal hyperplane. Consider Figure 5.27, which shows two possible separating hyperplanes and their associated margins. Before we get into the definition of margins, let's take an intuitive look at this figure. Both hyperplanes can correctly classify all the given data tuples. Intuitively, however, we expect the hyperplane with the larger margin to be more accurate at classifying future data tuples than the hyperplane with the smaller margin. This is why (during the learning or training phase) the SVM searches for the hyperplane with the largest margin, that is, the maximum marginal hyperplane (MMH). The associated margin gives the largest separation between classes.



(a)

Figure 5.27: SVM With Possible Separating hyperplanes

Getting to an informal definition of margin, we can say that the shortest distance from a hyperplane to one side of its margin is equal to the shortest distance from the hyperplane to the other side of its margin, where the "sides" of the margin are parallel to the hyperplane. When dealing with the MMH, this distance is, in fact, the shortest distance from the MMH to the closest training tuple of either class. A separating hyperplane can be written as $W.X + b = 0$ Where W is a weight vector, namely, W $=w_1, w_2, \ldots, w_n$; n is the number of attributes; and b is a scalar, often referred to as a bias. To aid in visualization, let's consider two input attributes, A1 and A2, as in Figure 5.27 (b). Training tuples are 2-D (e.g., X = (x1, x2)), where x1 and x2 are the values of attributes A1 and A2, respectively, for X. If we think of b as an additional weight, w0, we can rewrite the previous equation as $w_0 + w_1x_1 + w_2x_2 = 0$

Thus, any point that lies above the separating hyperplane satisfies

$$w_0 + w_1x_1 + w_2x_2 > 0 \tag{5.19}$$

Similarly, any point that lies below the separating hyperplane satisfies

$$w_0 + w_1x_1 + w_2x_2 < 0 \tag{5.20}$$

The weights can be adjusted so that the hyperplanes defining the "sides" of the margin can be written as

$$H_1 : w_0 + w_1 x_1 + w_2 x_2 >= 1 \text{ for } y_i = +1, \tag{5.21}$$

$$H_2 : w_0 + w_1 x_1 + w_2 x_2 =< -1 \text{ for } y_i = -1 \tag{5.22}$$

That is, any tuple that falls on or above H1 belongs to class +1, and any tuple that falls on or below H2 belongs to class -1. Combining the two inequalities:

$$y_i(w_0 + w_1 x_1 + w_2 x_2) >= 1 \text{ for all i.} \tag{5.23}$$

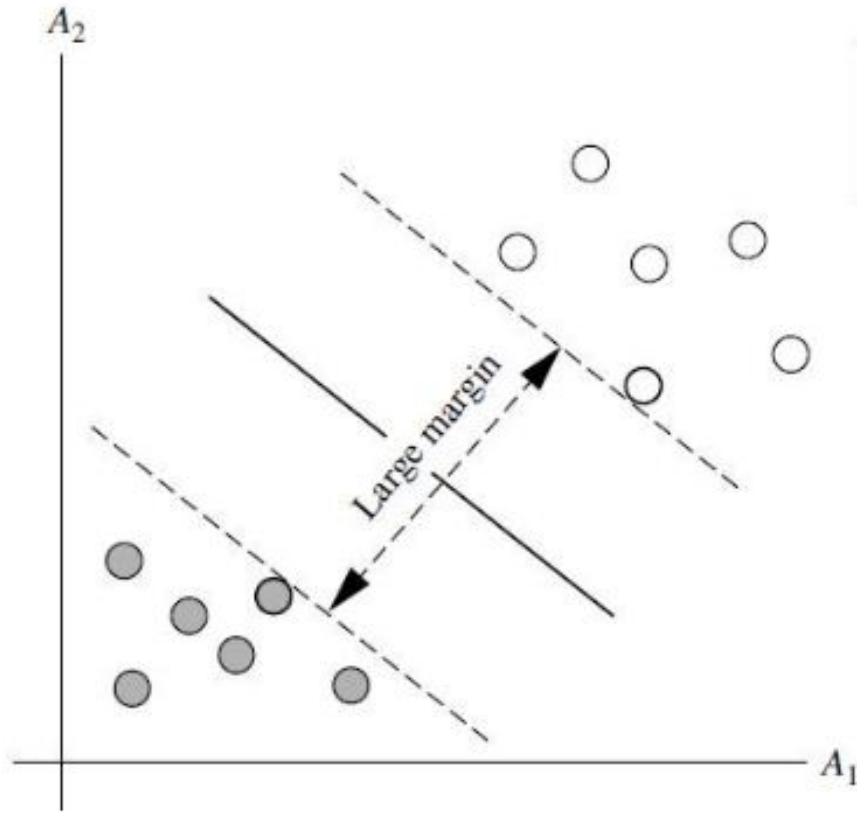
Any training tuples that fall on hyperplanes H1 or H2 (i.e., the "sides" defining the margin) satisfy the previous equation and are called support vectors. That is, they are equally close to the (separating) MMH. In 5.28, the support vectors are shown encircled with a thicker border. Essentially, the support vectors are the most difficult tuples to classify and give the most information regarding classification.

From this, we can obtain a formula for the size of the maximal margin. The distance from the separating hyperplane to any point on H1 is $1/|(|W|)|$, where $|(|W|)|$ is the Euclidean norm of W, that is, $\sqrt{W.W2}$. By definition, this is equal to the distance from any point on H2 to the separating hyperplane. Therefore, the maximal margin is $2/|(|W|)|$. Using some "fancy math tricks," we can rewrite the last equation so that it becomes what is known as a constrained (convex) quadratic optimization problem. Note that the tricks involve rewriting the last equation using a Lagrangian formulation and then solving for the solution using Karush-Kuhn-Tucker (KKT) conditions. If the data are small (say, less than 2000 training tuples), any optimization software package for solving constrained convex quadratic problems can then be used to find the support vectors and MMH. For larger data, special and more efficient algorithms for training SVMs can be used instead. Once we've found the support vectors and MMH (note that the support vectors define the MMH!), we have a trained support vector machine. The MMH is a linear class boundary, and so the corresponding SVM can be used to classify linearly separable data. We refer to such a trained SVM as a linear SVM.

Based on the Lagrangian formulation mentioned before, the MMH can be rewritten as the decision boundary;

$$d(X^T) = \sum_{i=1}^{l} y_i \alpha_i X_i X^T + b_0 \tag{5.24}$$

Where y_i is the class label of support vector X_i , X^T is a test tuple; α_i and b_0 are numeric parameters that were determined automatically by the optimization or SVM algorithm noted before; and l is the number of support vectors. α_i are Lagrangian



(a)

Figure 5.28: Higlight Support Vectors

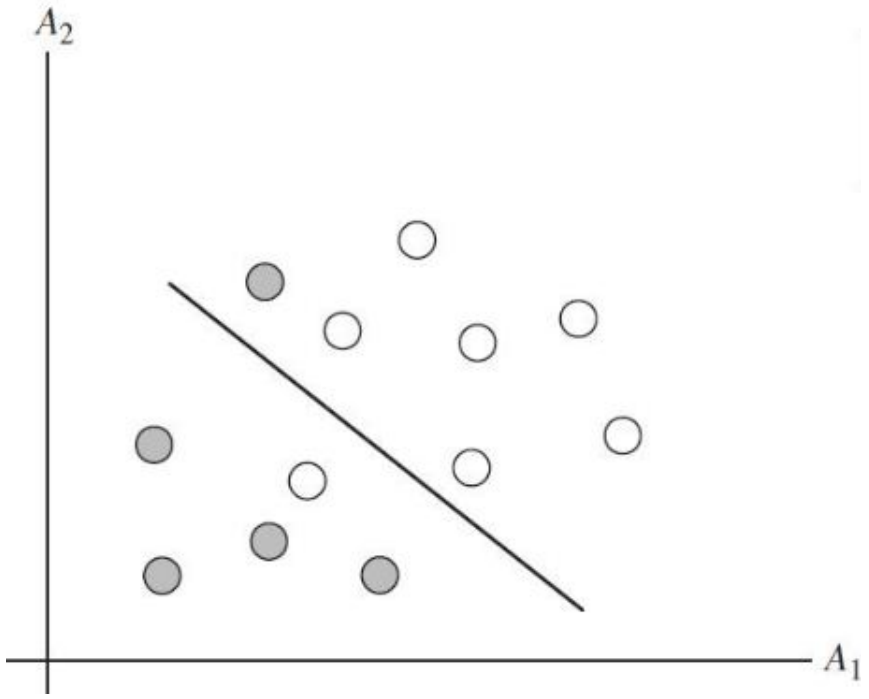
multipliers. For linearly separable data, the support vectors are a subset of the actual training tuples (although there will be a slight twist regarding this when dealing with non linearly separable data, as we shall see in the following).

Given a test tuple, X^T, we plug it into last equation and then check to see the sign of the result. This tells us on which side of the hyperplane the test tuple falls. If the sign is positive, then X^T falls on or above the MMH, and so the SVM predicts that X^T belongs to class +1 (representing buys computer = yes, in our case). If the sign is negative, then X^T falls on or below the MMH and the class prediction is -1 (representing buys computer = no). Notice that the Lagrangian formulation of our problem from the last equation contains a dot product between support vector X_i and test tuple X^T. This will prove very useful for finding the MMH and support vectors for the case when the given data are non linearly separable, as described further in the next section. Before we move on to the nonlinear case, there are two more important things to note. The complexity of the learned classifier is characterized by the number of support vectors rather than the dimensionality of the data. Hence, SVMs tend to be less prone to overfitting than some other methods. The support vectors are the essential or critical training tuples—they lie closest to the decision boundary (MMH). If all other training tuples were removed and training were repeated, the same separating hyperplane would

be found. Furthermore, the number of support vectors found can be used to compute an (upper) bound on the expected error rate of the SVM classifier, which is independent of the data dimensionality. An SVM with a small number of support vectors can have good generalization, even when the dimensionality of the data is high.

Linear In-separability Of The Data

The approach described for linear SVMs can be extended to create nonlinear SVMs for the classification of linearly inseparable data (also called nonlinearly separable data, or nonlinear data for short). Such SVMs are capable of finding nonlinear decision boundaries (nonlinear hypersurfaces) in input space. We obtain a nonlinear SVM by extending the approach for linear SVMs as follows. There are two main steps. In the first step, we transform the original input data into a higher dimensional space using a nonlinear mapping. Several common nonlinear mappings can be used in this step, as we will further describe next. Once the data have been transformed into the new higher space, the second step searches for a linear separating hyperplane in the new space. We again end up with a quadratic optimization problem that can be solved using the linear SVM formulation. The maximal marginal hyperplane found in the new space corresponds to a nonlinear separating hypersurface in the original space.



(a)

Figure 5.29: Linear In-separable Data

However, there are some problems. First, how do we choose the nonlinear mapping to a higher dimensional space? Second, the computation involved will be costly. Referring to the last equation, for the classification of a test tuple, X^T. Given the test tuple, we have to compute its dot product with every one of the support vectors. In training, we

have to compute a similar dot product several times in order to find the MMH. This is especially expensive. Hence, the dot product computation required is very heavy and costly.

We can use another math trick. It so happens that in solving the quadratic optimization problem of the linear SVM (i.e., when searching for a linear SVM in the new higher dimensional space), the training tuples appear only in the form of dot products, $\phi x_i)$ $.\phi x_j$, where $\phi x_i)$ is simply the nonlinear mapping function applied to transform the training tuples. Instead of computing the dot product on the transformed data tuples, it turns out that it is mathematically equivalent to instead apply a kernel function, $K(x_i, x_j)$ to the original input data. That is:

$$K(x_i, x_j) = \phi x_i \phi x_j. \tag{5.25}$$

In other words, everywhere that $\phi x_i.\phi x_j$.appears in the training algorithm, we can replace it with $K(x_i, x_j)$. In this way, all calculations are made in the original input space, which is of potentially much lower dimensionality! We can safely avoid the mapping—it turns out that we don't even have to know what the mapping is! We will talk more later about what kinds of functions can be used as kernel functions for this problem.

After applying this trick, we can then proceed to find a maximal separating hyperplane.

Properties of the kinds of kernel functions that could be used to replace the dot product scenario just described have been studied. Three admissible kernel functions are:

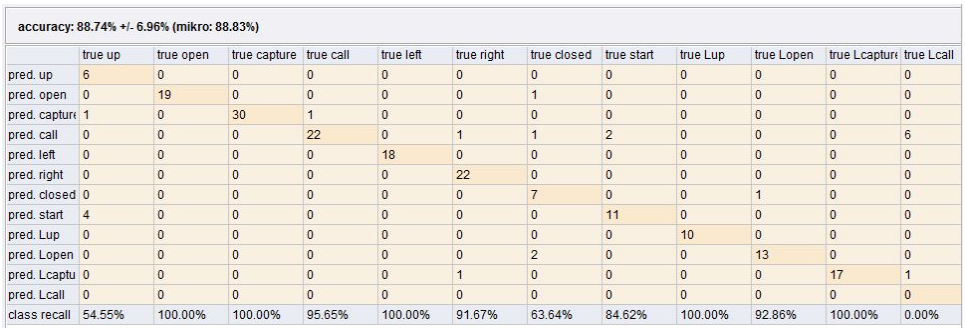
- Polynomial kernel of degree h : $K(x_i, x_j) = (x_i.x_j + 1)^h$.
- Gaussian radial basis function kernel: $K(x_i, x_j) = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$.
- Sigmoid kernel: $K(x_i, x_j) = tanh(x_i.x_i - \delta)$.

Each of these results in a different nonlinear classifier in (the original) input space. Neural network aficionados will be interested to note that the resulting decision hyperplanes found for nonlinear SVMs are the same type as those found by other well-known neural network classifiers. For instance, an SVM with a Gaussian radial basis function (RBF) gives the same decision hyperplane as a type of neural network known as a radial basis function network. An SVM with a sigmoid kernel is equivalent to a simple two-layer neural network known as a multilayer perceptron (with no hidden layers).

There are no golden rules for determining which admissible kernel will result in the most accurate SVM. In practice, the kernel chosen does not generally make a large difference in resulting accuracy. SVM training always finds a global solution, unlike neural networks, such as back propagation.

So far, we have described linear and nonlinear SVMs for binary (i.e., two-class) classification. SVM classifiers can be combined for the multiclass case. A major research goal regarding SVMs is to improve the speed in training and testing so that SVMs may become a more feasible option for very large data sets (e.g., millions of support vectors). Other issues include determining the best kernel for a given data set and finding more efficient methods for the multiclass case.

SVM has the following results on our datasets. For Binary Representation Algorithm, it gives the best results according to the following results that describe the accuracy of ever static algorithm.

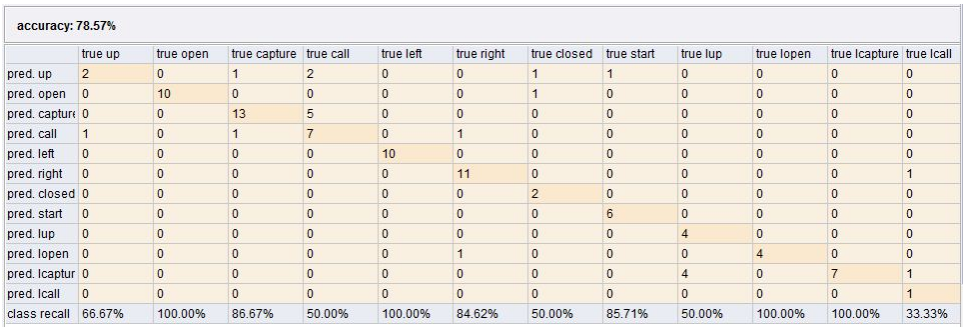


accuracy: 88.74% +/- 6.96% (mikro: 88.83%)

	true up	true open	true capture	true call	true left	true right	true closed	true start	true Lup	true Lopen	true Lcapture	true Lcall
pred. up	6	0	0	0	0	0	0	0	0	0	0	0
pred. open	0	19	0	0	0	0	1	0	0	0	0	0
pred. capture	1	0	30	1	0	0	0	0	0	0	0	0
pred. call	0	0	0	22	0	1	1	2	0	0	0	6
pred. left	0	0	0	0	18	0	0	0	0	0	0	0
pred. right	0	0	0	0	0	22	0	0	0	0	0	0
pred. closed	0	0	0	0	0	0	7	0	0	1	0	0
pred. start	4	0	0	0	0	0	0	11	0	0	0	0
pred. Lup	0	0	0	0	0	0	0	0	10	0	0	0
pred. Lopen	0	0	0	0	0	0	2	0	0	13	0	0
pred. Lcaptu	0	0	0	0	0	1	0	0	0	0	17	1
pred. Lcall	0	0	0	0	0	0	0	0	0	0	0	0
class recall	54.55%	100.00%	100.00%	95.65%	100.00%	91.67%	63.64%	84.62%	100.00%	92.86%	100.00%	0.00%

(a)

Figure 5.30: Binary Based Approach Results Using x-validation with k=10



accuracy: 78.57%

	true up	true open	true capture	true call	true left	true right	true closed	true start	true lup	true lopen	true lcapture	true lcall
pred. up	2	0	1	2	0	0	1	1	0	0	0	0
pred. open	0	10	0	0	0	0	1	0	0	0	0	0
pred. capture	0	0	13	5	0	0	0	0	0	0	0	0
pred. call	1	0	1	7	0	1	0	0	0	0	0	0
pred. left	0	0	0	0	10	0	0	0	0	0	0	0
pred. right	0	0	0	0	0	11	0	0	0	0	0	1
pred. closed	0	0	0	0	0	0	2	0	0	0	0	0
pred. start	0	0	0	0	0	0	0	6	0	0	0	0
pred. lup	0	0	0	0	0	0	0	0	4	0	0	0
pred. lopen	0	0	0	0	0	1	0	0	0	4	0	0
pred. lcaptur	0	0	0	0	0	0	0	0	4	0	7	1
pred. lcall	0	0	0	0	0	0	0	0	0	0	0	1
class recall	66.67%	100.00%	86.67%	50.00%	100.00%	84.62%	50.00%	85.71%	50.00%	100.00%	100.00%	33.33%

(a)

Figure 5.31: Feature Based Approach Results Using x-validation with k=10

Naïve Base

Not Yet

KNN

The k-nearest-neighbor method [**classifications**] was first described in the early 1950s. The method is labor intensive when given large training sets, and did not gain popularity until the 1960s when increased computing power became available. It has since been widely used in the area of pattern recognition.

Nearest-neighbor classifiers are based on learning by analogy, that is, by comparing a given test tuple with training tuples that are similar to it. The training tuples are described by n attributes. Each tuple represents a point in an n-dimensional space. In this way, all the training tuples are stored in an n-dimensional pattern space.When given an unknown tuple, a k-nearest-neighbor classifier searches the pattern space for the k

training tuples that are closest to the unknown tuple. These k training tuples are the k "nearest neighbors" of the unknown tuple.

"Closeness" is defined in terms of a distance metric, such as Euclidean distance. The Euclidean distance between two points or tuples, say, $X1 = (x_{11}, x_{12}, ..., x_{1n})$ and $X2 = (x_{21}, x_{22}, ..., x_{2n})$, is

$$dist(x_1, x_2) = \sqrt{\sum_{i=1}^{n} (x_{1i} - x_{2i})^2} \tag{5.26}$$

For k-nearest-neighbor classification, the unknown tuple is assigned the most common class among its k-nearest neighbors. When k = 1, the unknown tuple is assigned the class of the training tuple that is closest to it in pattern space. Nearest-neighbor classifiers can also be used for numeric prediction, that is, to return a real-valued prediction for a given unknown tuple. In this case, the classifier returns the average value of the real-valued labels associated with the k-nearest neighbors of the unknown tuple.

The previous discussion assumes that the attributes used to describe the tuples are all numeric. For nominal attributes, a simple method is to compare the corresponding value of the attribute in tuple X1 with that in tuple X2. If the two are identical (tuples X1 and X2 both have the color blue), then the difference between the two is taken as 0. If the two are different (tuple X1 is blue but tuple X2 is red), then the difference is considered to be 1. Other methods may incorporate more sophisticated schemes for differential grading (where a larger difference score is assigned, say, for blue and white than for blue and black).

"How can I determine a good value for k, the number of neighbors?" This can be determined experimentally. Starting with k = 1, we use a test set to estimate the error rate of the classifier. This process can be repeated each time by incrementing k to allow for one more neighbor. The k value that gives the minimum error rate may be selected. In general, the larger the number of training tuples, the larger the value of k will be (so that classification and numeric prediction decisions can be based on a larger portion of the stored tuples). As the number of training tuples approaches infinity and k = 1, the error rate can be no worse than twice the Bayes error rate (the latter being the theoretical minimum). If k also approaches infinity, the error rate approaches the Bayes error rate.

Nearest-neighbor classifiers can be extremely slow when classifying test tuples. If D is a training database of |D| tuples and k = 1, then O(|D|) comparisons are required to classify a given test tuple. By presorting and arranging the stored tuples into search trees, the number of comparisons can be reduced to O(log|D|). Parallel implementation can reduce the running time to a constant, that is, O(1), which is independent of |D|. Other techniques to speed up classification time include the use of partial distance calculations and editing the stored tuples. In the partial distance method, we compute the distance based on a subset of the n attributes. If this distance exceeds a threshold, then further computation for the given stored tuple is halted, and the process moves on to the next stored tuple. The editing method removes training tuples that prove useless. This method

is also referred to as pruning or condensing because it reduces the total number of tuples stored.

K-NN has the following results on our datasets. For Binary Representation Algorithm, it gives the best results according to the following results that describe the accuracy of ever static algorithm.

accuracy: 96.42% +/- 3.24% (mikro: 96.45%)

	true up	true open	true capture	true call	true left	true right	true closed	true start	true Lup	true Lopen	true Lcapture	true Lcall
pred. up	9	0	0	0	0	0	0	1	0	0	0	0
pred. open	0	19	0	0	0	0	0	0	0	0	0	0
pred. capture	0	0	30	0	0	0	0	0	0	0	0	0
pred. call	0	0	0	23	0	0	0	1	0	0	0	1
pred. left	1	0	0	0	18	0	0	0	0	0	0	0
pred. right	0	0	0	0	0	24	0	0	0	0	0	0
pred. closed	0	0	0	0	0	0	10	0	0	1	0	0
pred. start	1	0	0	0	0	0	1	11	0	0	0	0
pred. Lup	0	0	0	0	0	0	0	0	10	0	0	0
pred. Lopen	0	0	0	0	0	0	0	0	0	13	0	0
pred. Lcaptu	0	0	0	0	0	0	0	0	0	0	17	0
pred. Lcall	0	0	0	0	0	0	0	0	0	0	0	6
class recall	81.82%	100.00%	100.00%	100.00%	100.00%	100.00%	90.91%	84.62%	100.00%	92.86%	100.00%	85.71%

(a)

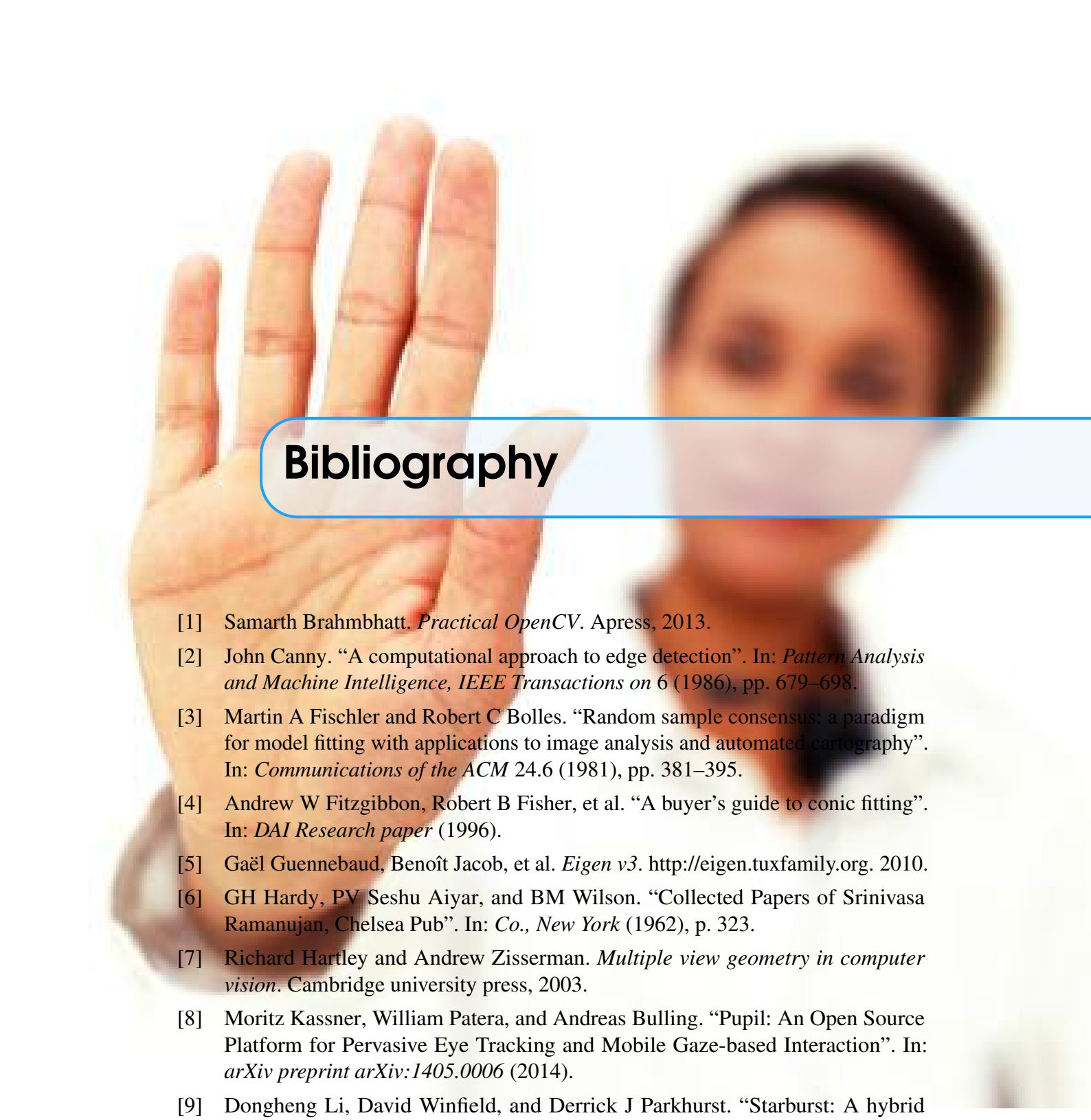
Figure 5.32: Binary Based Approach Results Using x-validation with k=10

accuracy: 87.76% +/- 7.69% (mikro: 87.82%)

	true up	true open	true capture	true call	true left	true right	true closed	true start	true lup	true lopen	true lcapture	true lcall
pred. up	8	0	1	0	0	2	1	1	0	0	0	0
pred. open	0	17	0	0	0	0	1	0	0	0	0	0
pred. capture	0	0	27	5	0	0	0	1	0	0	0	0
pred. call	2	0	2	17	0	1	0	0	0	0	0	0
pred. left	0	0	0	1	18	0	0	0	0	0	0	0
pred. right	0	0	0	0	0	21	0	0	0	0	0	0
pred. closed	0	1	0	0	0	0	9	0	0	0	0	0
pred. start	1	1	0	0	0	0	0	11	0	0	0	0
pred. lup	0	0	0	0	0	0	0	0	10	0	1	0
pred. lopen	0	0	0	0	0	0	0	0	0	14	0	0
pred. lcapture	0	0	0	0	0	0	0	0	0	0	16	2
pred. lcall	0	0	0	0	0	0	0	0	0	0	0	5
class recall	72.73%	89.47%	90.00%	73.91%	100.00%	87.50%	81.82%	84.62%	100.00%	100.00%	94.12%	71.43%

(a)

Figure 5.33: Feature Based Approach Results Using x-validation with k=10



Bibliography

[1] Samarth Brahmbhatt. *Practical OpenCV*. Apress, 2013.

[2] John Canny. "A computational approach to edge detection". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 6 (1986), pp. 679–698.

[3] Martin A Fischler and Robert C Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". In: *Communications of the ACM* 24.6 (1981), pp. 381–395.

[4] Andrew W Fitzgibbon, Robert B Fisher, et al. "A buyer's guide to conic fitting". In: *DAI Research paper* (1996).

[5] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[6] GH Hardy, PV Seshu Aiyar, and BM Wilson. "Collected Papers of Srinivasa Ramanujan, Chelsea Pub". In: *Co., New York* (1962), p. 323.

[7] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[8] Moritz Kassner, William Patera, and Andreas Bulling. "Pupil: An Open Source Platform for Pervasive Eye Tracking and Mobile Gaze-based Interaction". In: *arXiv preprint arXiv:1405.0006* (2014).

[9] Dongheng Li, David Winfield, and Derrick J Parkhurst. "Starburst: A hybrid algorithm for video-based eye tracking combining feature-based and model-based approaches". In: *Computer Vision and Pattern Recognition-Workshops, 2005. CVPR Workshops. IEEE Computer Society Conference on*. IEEE. 2005, pp. 79–79.

[10] Dave M Stampe. "Heuristic filtering and reliable calibration methods for video-based pupil-tracking systems". In: *Behavior Research Methods, Instruments, & Computers* 25.2 (1993), pp. 137–142.

[11] Satoshi Suzuki et al. "Topological structural analysis of digitized binary images by border following". In: *Computer Vision, Graphics, and Image Processing* 30.1 (1985), pp. 32–46.

[12] Lech Świrski, Andreas Bulling, and Neil Dodgson. “Robust real-time pupil tracking in highly off-axis images”. In: *Proceedings of the Symposium on Eye Tracking Research and Applications*. ACM. 2012, pp. 173–176.

[13] Paul Viola and Michael Jones. “Rapid object detection using a boosted cascade of simple features”. In: *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*. Vol. 1. IEEE. 2001, pp. I–511.

[14] Danjie Zhu, Steven T Moore, and Theodore Raphan. “Robust pupil center detection using a curvature algorithm”. In: *Computer methods and programs in biomedicine* 59.3 (1999), pp. 145–157.

Index

I

K

L

M

N

O

P

R

S