Institute of NeuroInformatics
Uni Irchel Zurich

IMM – Informatics and Mathematical modelling
Technical University of Denmark

Active shape models on Silicon retina

C O G A I N
Silicon retina project

Supervisor:

Tobi Delbruck

Bjarne K. Ersbøll

Project done by:
Alexander Martin Tureczek

Table of contents

1 Introduction and problem statement

This report contains the documentation of the work done in collaboration between DTU and Uni Zürich in the fall of 07 and spring of 08.

Having seen active appearance models (AAM) being very successful on frame based images, this report will look into the possibility to use them on event based data, such as coming from the silicon retina camera.

The AAM will be build on a set of existing images of faces, through annotation of important landmarks in a face shape we will create a mean shape of these different shapes. This shape will be used as an initial estimate of the face.

We will use 2 LED lights mounted on a pair of glasses to identify where the face is situated in the event stream. Then from a variable number of events we will see if it is possible to use the LED plus the events to deform the face to fit the data better. Depending on the success of the deformation AAM will also be used to model a mouth in the face.

It is the hope that this project will lead to a filter implementation useful in the jAER project on sourceForge.com. Firstly a proof of concept will be made in MatLab to see if the data are useful, and the idea is realizable. Then a large part of the project will be to gain the knowledge to implement the MatLab code into JAVA.

Knowing that descriptive statistics, like the median will become an important part of estimating the position of different landmark and LED, an implementation of the very efficient sorting algorithm Heap Sort will be made. In a given java filter implementation, sorting of the events will be the most expensive procedure, being able to do this efficiently is of prime importance. Thus an interesting study will be the computational effect required by the filter will be made as well.

This gives three main focuses of this project; modeling a face, e.g. be able to detect it in an event stream, and also model a mouth from this event stream. While thirdly implementing the proof of concept into a JAVA filter. A side project will be to analyze the run time performance of the jAER filter implementation.

2 Theory

The models are done using Procrustean shape analysis fitted to a shape, while the techniques from T.F Cootes' Statistical models of appearance is going to be the basis of our project.

2.1 Procrustean mean shape

The procrustean mean shape is used to calculate a mean shape of the objects in question. It is built on known positions of landmarks from a training set. The trainings shapes are aligned to each other see figure 2.1, and then calculating the average shape from the difference of corresponding landmarks.

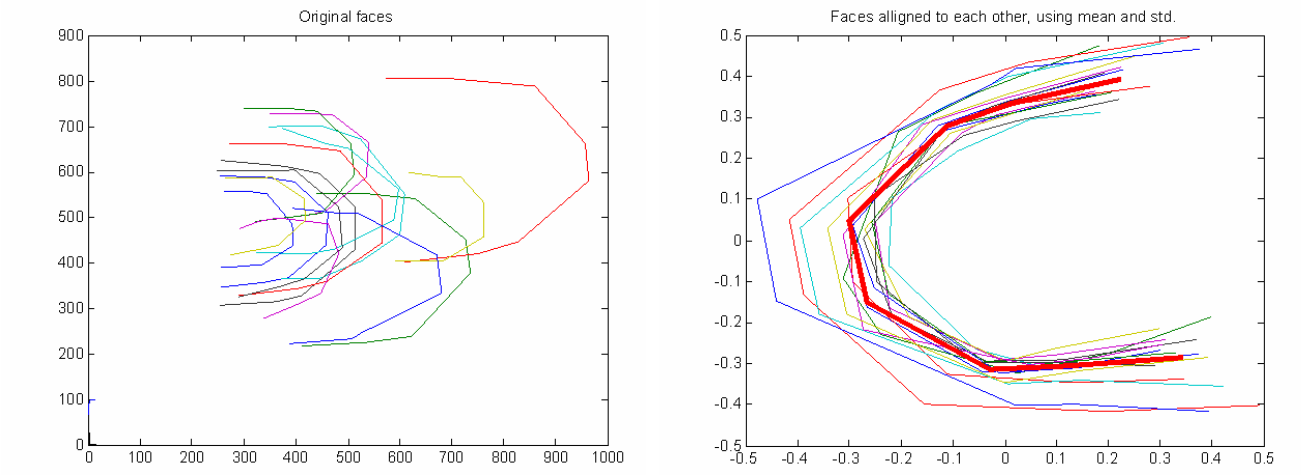

Figure 2.1 – (left) the original faces with their size difference and positional difference. (Right) Faces aligned with mean and standardized to the mean shape showing the difference in the shapes regardless of size. The fat red line is the mean shape of all shapes used.

Procrustes is reducing the sum of squares between the points, this is done by the following:

$$S = \sum_{i=1}^{n} w_i w_i^* / (w_i^* w_i) = \sum_{i=1}^{n} z_i z_i^* \quad (1)$$

Where; $z = w_i / \| w_i \|$, i=1,…,n. This formula is calculating the complex sum of squares of the shapes w. where w* is the conjugate. The shapes are converted to complex as this makes it easier to compute in MatLab with a shape given in a vector a+ib rather than two vectors of x and y. The matrix S is then complex sum of squares. The largest eigenvector in the matrix S is the mean shape of the included shapes.

$$\hat{\mu} = \sup(eig(S)) \quad (2)$$

This shape is unique except in rotation. As the mean shape vector is rotated with respect to the original shapes we need to align the shapes to the mean shape to enable us to calculate the variation in the shapes.

$$w_i^p = w_i^* \hat{\mu} w_i / (w_i^* w_i) \tag{3}$$

This provides the rotation from the initial shapes to the mean shape calculated in (1). The resulting mean shape and alignment to this shape is shown figure 2.1 (right), while the initial condition is shown to the left. When the shapes are aligned we are able to calculate the variance in the shapes with respect to the mean shape.

$$\text{Var} = \frac{1}{n-1}\sum_{i=1}^{s}(x_i - \bar{x})(x_i - \bar{x})^T \tag{4}$$

n is the number of shapes, i=1,…,n this gives us a matrix of dispersions where the diagonal is the variance and other observations is the covariance between shapes. When calculating the variance we stop using the complex notation and create a (x, y) vector looking like (5)

$$w = \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ y_1 \\ \vdots \\ y_n \end{pmatrix} \tag{5}$$

2.2 Active shape

The Procrustean means shape and dispersion matrix are used in active shape modeling to define the model after which the objects are described. The idea is using; the mean shape and allowing for deformation via the dispersion matrix, it is possible to fit the mean shape to an object with the same features. E.g. a model of a face would then be deformed into a new face.

$$x \approx \bar{x} + \Phi b \tag{6}$$

(6) Shows the idea of the deformation. Having a mean shape $\bar{x}$ the adding some deformation to this shape in Φ. The deformation is controlled by the parameter vector b. Φ is a matrix consisting of eigenvectors found with PCA, and explaining more than some arbitrary threshold of the total variation in the data.

This method uses expert knowledge of objects to detect new object of the same type, but it cannot deform into a shape that wasn't in the trainings set, e.g. it's impossible to deform a square into a circle if this link hasn't been established already in our model.

This report will use as many eigenvectors as to describe approximately 95% of the total variation in the data.

3 Modeling the face

This section will be concerned with the modeling of a face using techniques from T. F. Cootes et al, on face shapes. We will develop the shape model that is going to become the face. Firstly we will identify where the face is situated in front of the camera.

3.1 Detection of the face in the retina image:

To make it easier to detect the face in the spiking image from the silicon retina, we mount 2 LED's on a par of glasses giving us a fix point all the time due to the flickering of the LED. This report will only describe the frequency of 2 kHz, but the MatLab program was tested on various different frequencies below 2 kHz, namely 1.5 kHz, 1 kHz 500 Hz, and 100 Hz..

3.1.1 Identifying the face with 1 LED:

Since the LED is set to a high frequency, it is hoped to be the dominant event in the event stream. Therefore we can for each 100 or 1000 events calculate the median in the x and y coordinate and get a very good indication of where the led was in that time interval. With modern sorting algorithms this sorting processes it doable in $O(n \cdot \ln(n))$ calculations.

Figure 3.1 below shows how the bar chart looks for a 100 event x-axis (Left) it is clearly seen that a median approach would give a good approximation to the position on the axis of the LED. Also in this way noise will have nothing to very little impact on the position of the LED. Figure 3.1 (Right) shows the same for the y-axis.

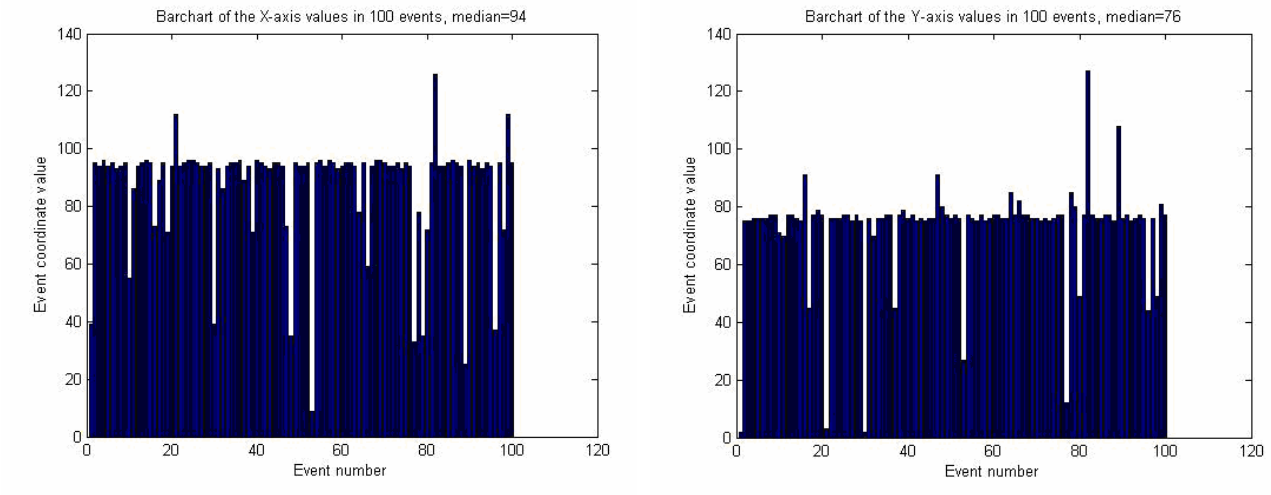

Figure 3.1 - Identifying the LED position using the median of the position on each axis. The coordinate/position is plotted as a function of the event number, we see from left and right, that there is one coordinate on both axis that is dominant. This coordinate is then used as our initial guess on the position of the LED.

In small time steps it is very robust, but if the whole image is analyzed at once this will not work, because then the median would include all information from the face moving around. Also using 1000 events defines sharply the median and hence the position of the LED. Ref. figure 3.3 to see the estimated positions of the LED.

3.1.2 Identifying the 2 LED:

As mentioned earlier we are using LEDs to identify the position of the face. Using 2 LEDs also gives us information of the angle/tilt of the face. To identify the two LEDs we use the median, because of its robustness towards extreme observations, or spread in data. But since we use two LEDs we are going to get an estimated median between the LEDs. To get an estimate on each of the two LEDs actual positions we partition the sorted data in a left (LED 1) and a right (LED 2) part, using the grand median as the dividing point. Now finding the median in each of the parts gives an initial estimate on the position of an LED. Figure 3.2 shows how the partitioning of the data is done.

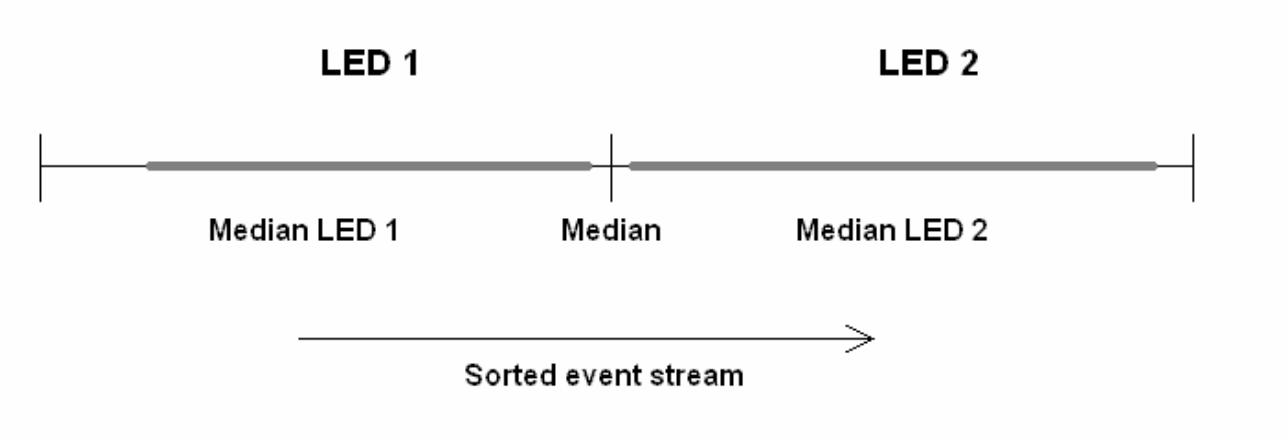

Figure 3.2 – the partitioning of the data via the grand median into smaller regions LED 1 and LED 2, making it easier to get a good estimate on the position of the LEDs.

To prevent the LED moving too much between iterations, we constrain the LED median calculation to only include a predefined number of pixels around the LED, making a box where the LED can move freely inside. The centre of the box is always set to be the estimate of the LED from the previous iteration.

The above mentioned technique for finding the LEDs is easy to use for the x axis where there is a clear distinction between the two LEDs, for the y axis it becomes a problem because often the y-coordinate of the LEDs will be very close but not identical.

We solve this by letting the LED wander around as described above, using just one estimate for both y-coordinate of the LED and then let it find its median independently, also calculating the median on each cell, LED1 and LED2.

After having identified the coordinates for both LEDs we have estimates of the position of the face and hence we have an idea of where to place our model. Figure 3.3 shows the LEDs (left) and the fitted face (Right)

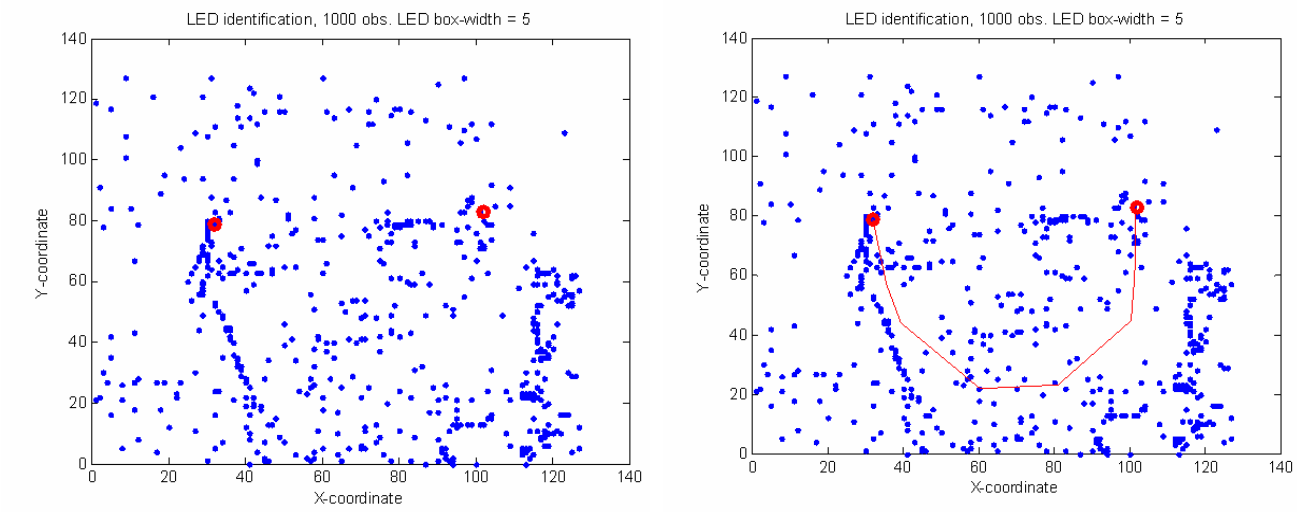

Figure 3.3 – left is seen how the LED are positioned in the data, no apparent face is visible, the vertical lines are cables from the LEDs that unfortunately moved during the making of the sequence. Right is imposed a scaled and rotated face making it easier to see where to expect a face, also making it possible to identify the mouth and nose of the face.

Experiments using LED with a frequency of 2 kHz showed that the LED does not make a big enough cluster for the estimate to be stable when using 1000 observations. Exactly 1000 observations has been the default sample size of events during this project, though this is configurable by the user, more data didn't change this tendency. Figure 3.4 show a LED1 and a LED2 estimates, it is clearly seen that the LED1 is not a stable estimate like the LED2. Also from figure 3.10 it is seen that the LED may wander of.

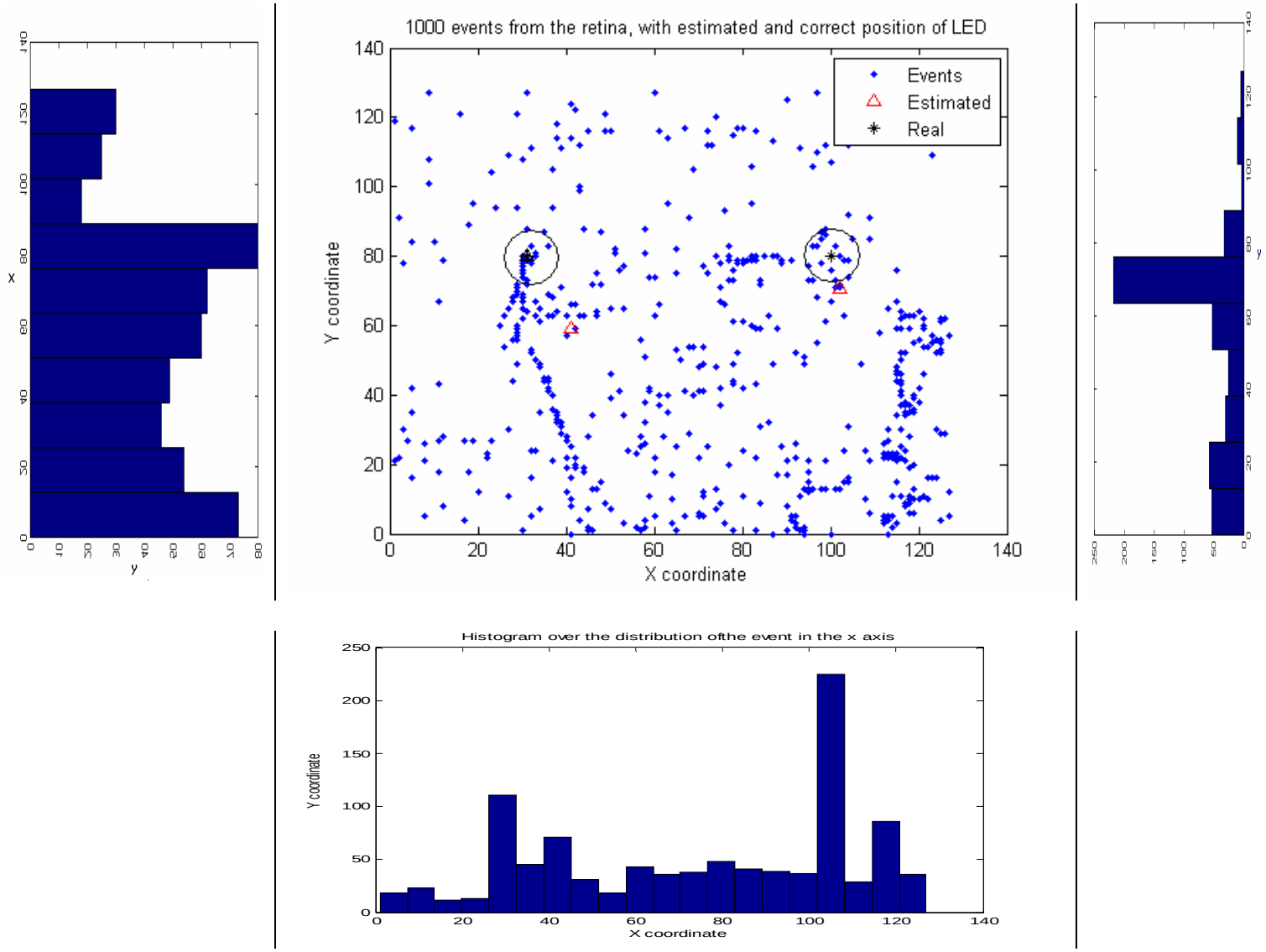

Figure 3.4 – the first 1000 observations of two LED with 2 kHz. It is seen that the left LED is considerably harder to estimate than the right LED. Especially the left histogram shows the tendency of either one of the LEDs, this spread of events makes the LED more likable to wander of.

The phenomena shown in figure 3.4, was experienced often, meaning that the estimated LED would jump around in the image, disregarding the LED. Though the LED is blinking with high frequency, here 2 kHz, it did not provide an all too stabile estimate. We reduced the amount of instability by restricting the movement of the LED at each iteration but as figure 3.11 shows movement in the image, coming from a small cord was enough to get the LED to wander of even when the allowed movement was small. To really overcome this problem a more stabile estimate of the LED has to be devised, which has not been done in this project.

3.1.3 Computational effort regarding calculation of median

Since the LED is set to a high frequency, it will be the dominant event series in the event stream. Therefore we can for each 100 or 1000 events calculate the median in the x and y coordinate and get an indication of where the LED was in that time interval. With modern sorting algorithms this

sorting process for finding the median has a worst case running time in $O(n \cdot \ln(n))$, this is worst case running time for the Heap Sort algorithm implemented in the Java filter.

3.2 Annotating faces from images:

Initially the focus was on getting the code working in MatLab with the annotation of points and construction of the model of a face. The face was set to consist of 8 landmarks, the intention is to make the code robust such that an annotated face with more landmarks can be used instead. When annotating one has to taking into consideration the distance between the landmarks. Landmarks cannot be situated to closely else it will be possible for data to be used to calculate updates for different landmarks.

The annotation of points is shown in figure 3.4, only the outline of the face was annotated, while the mouth is annotated in chap. 5 *modeling the mouth*. The intention of this model has been to keep it simple and thus try to model a face with as few landmarks as possible to still get a reasonable model of a face. This will keep the filter as cheap as possible.

As mentioned the face consists of 8 landmarks, and the mouth will be consisting of 6 landmarks, also seen in figure 3.4.

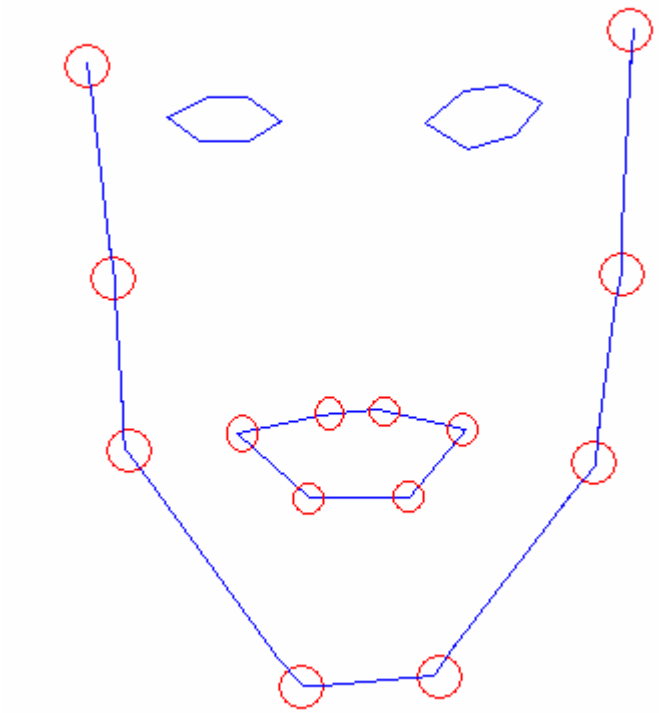

Figure 3.4 – Landmarks on a face used to construct our model. The eyes and lips are not included in the initial modeling process. On each cheek there is an additional annotation placed right below the ear, the top annotation on the cheek is placed where the ear ends

3.3 The Choice of faces:

The annotation of the face, is done using MatLab and photos of people. Initially the model will be built on faces looking into the camera. Both eyes and the mouth should be visible, putting a constraint that the model only fits to faces from front plus some small angle to the left/right and up/down. The images used were not standardized to look particular, they were taken from a series of different portrait photos. This was done in the hope that these photos would bring in more variation used for modeling different face shapes when a face is moving. This could also have the draw back that there may be introduced variation that cannot be accounted for, and thus will bias the model.

The preferable approach, would be to use two images of each person, one with closed mouth and one with open (like when talking) as to be able to create some shape variation in the mouth making it able to fit to a talking person. Ideally the model would be able to imitate a person during a conversation/talk. This is however only in the very optimum case this would be reachable for this project. Using two pictures of each person has not been tried in this project.

3.4 Creating the first meanshape:

To make sure the code works and that the theory has been applied correctly, we make an initial rough model, making sure we get the expected results. We build a model from 14 images, with each 8 annotations in complex notation, (making it possible to express the shape coordinates in one vector.)

Figure 3.5 shows two of the images chosen to generate the model. It is seen that both individuals are looking directly into the camera, (left) has his head in a vertical, “normal” position, and his mouth closed. While (right) has it tilted head position and open mouth. In this model we are not going to try and model mouth variation, but we will capture some variation due to different head positions. The images from figure 3.5 are representative of the 14 images used in this test. They all require that the person is looking into the camera, and the head is up right or in a tilted position like in 3.5 (right).

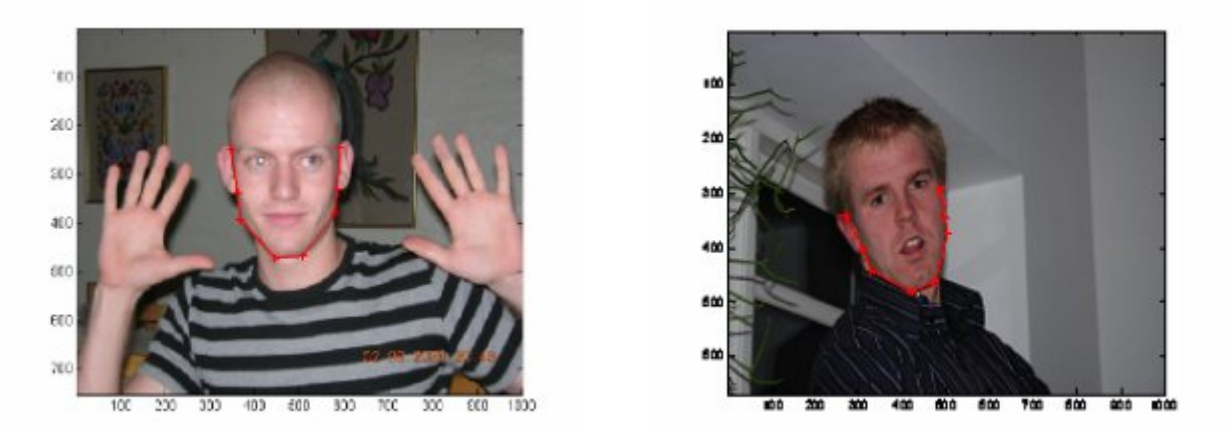

Figure 3.5 – Annotating a face outline. (Left) Face pointing towards camera whole face visible, mouth closed. (Right) Face pointing towards camera tilted and not all face contours are visible, mouth open. These serve to illustrate the difference in the trainings set, making the model more robust when people are moving.

Also the annotations are only approximate, as they have been annotated by hand and hence there will be induced some variation into the data from the person annotating the images. We will not try to take into account the variation induced by the person annotation the images.

After having annotated the shapes in the 14 images, we mean center and calculate the largest Eigen value. The results are shown in figure 3.6 (left) the fat red line in the model is the mean shape and it is clearly seen how the other shapes are varying around this shape. We even see the tilting of the different faces. Calculating the variation in the shapes, and decomposing the variation into eigenvectors gives us a clear view of how this variation is distributed over the 14 dimensions. Figure 3.6 (middle) shows in bars how much each principal component (PC) describes of the variation, while the graph shows the cumulated variance. It is seen that the first three/four PC's are describing roughly the most of the total variation in the data. To retain approx 95% of the variation we need three PC's

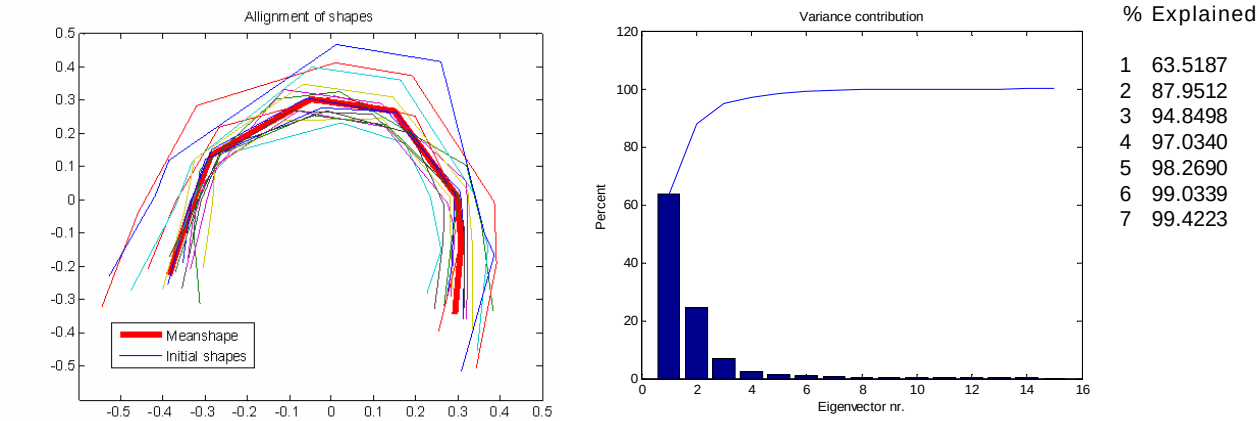

Figure 3.6 – (Left) alignment to the mean shape. (middle, bars) The explained variation for each eigenvector, (Right) the cumulated explained variance. This variance plot shows how fast we are able to capture approx 95% of the total variation in the data.

Plotting the first three eigenvectors and adding ±3 standard deviations ($\sqrt{\lambda}$) will show us what variation the three largest components are explaining, and how this will influence the model deformation. The plot is shown in figure 3.7

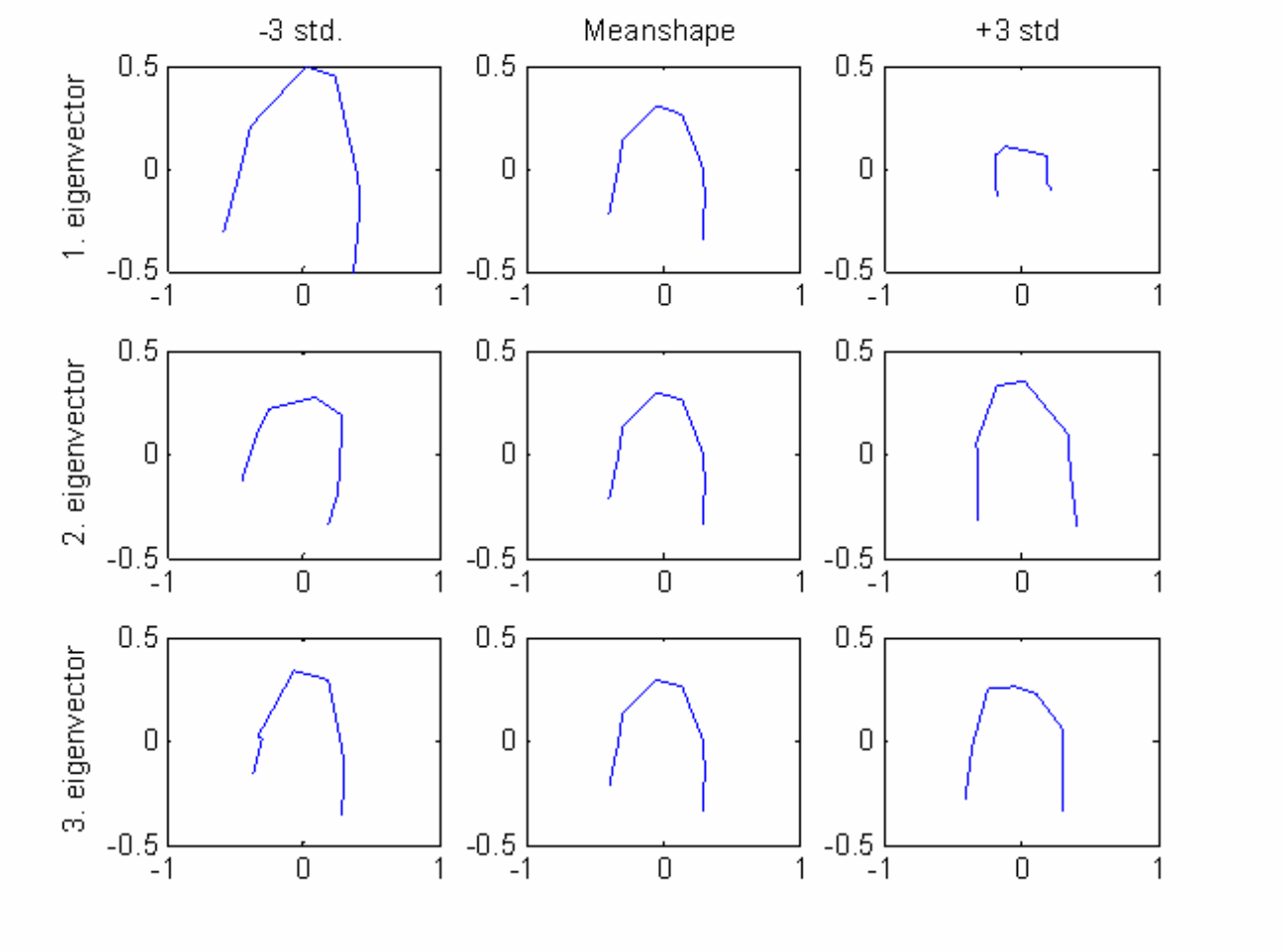

Figure 3.7 – The influence of the three largest eigenvectors on the mean shape.

We see from figure 3.7, that eigenvector 1 influences the size of the shape, and even also some deformation from a clearly rounded curve to almost being an open box with +3std. eigenvector 2 looks like it is controlling the rotation of the face, and a very small deformation of the jaw. Finally vector 3, is doing significant deformation to the jaw, almost as if the subjects were not looking directly into the camera.

3.5 Fitting the model to the face seen in the retina:

Using the LED as initial guess and to initialize the face model, setting the top of the model (highest landmark) at the points where the LED is seen, and then pull the model to fit the face when movement is detected. This means checking for each landmark in the mean shape if there is a difference between the mean shape landmark and the closest event in the retina. If so then we pull the shape by estimating the parameters from

$$x \approx \bar{x} + \Phi b \tag{7}$$

Using the knowledge of the difference between the model and the estimated landmarks from the silicon retina we can rewrite (7) into:

$$b = \Phi^T (x - \bar{x}) \tag{8}$$

Making us able to estimate the parameters and finally giving us a face fit to the retina event data.

3.6 Rotating the model to up-right position

The mean shape is the largest eigenvector and as such unique till an arbitrary rotation. This means every time the model is created the face is tilted with some angle. We want the initial guess of the face to be at the normal position (up right). As we believe this is the most plausible position for an initial guess, the problem is shown in figure 3.8 below

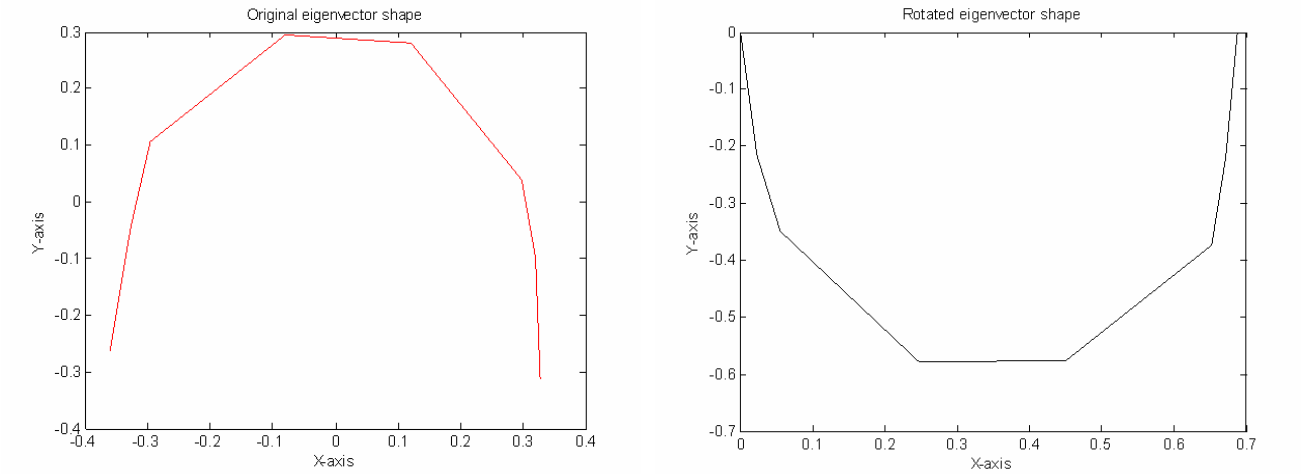

Figure 3.8 - Left is shown the original eigenvector shape, this is just an arbitrary rotation from the eigenvector properties, and the goal is to rotate the shape into the right figure. This will give us the initial estimate of the face before beginning to deform it.

We start by doing a translation such the upper left landmark is centered in (0, 0). The rotation of the face is determined from the angle between the upper left (UL) and upper right (UR) landmark, this angle has to finally become zero, resulting in figure 3.8 right. We calculate the angle between UL and UR using:

$$\theta = \arctan\left(\frac{y}{x}\right) \tag{9}$$

We also calculate the angle to every other landmark and subtract (9) from them. Remembering that there is 2 solutions for every result in (9). This does not pose a problem in the initial alignment phase, but when updating the model it does, so this has to be taken care of.

During the rotation we also calculate the distance between the UL and UR this is later used to calculate the scaling of the face. Though the model initial rotation and the later adaptation to the LED positions could be done in one code the initial rotation is implemented in the rotation.m and the later fit to the LED positions is done in the rotator.m file.

3.7 Rotating the model to follow the LED positions

The principle is the same as when rotating the face to initial position. Only now we don't have that the angle between UL and UR (both LEDs) is 0. This means that we have to calculate the angle again of every point using (9), where y/x gives us the tilt/slope of the present LED positions. This is then compared with the slope of the previous LED positions (last iteration.). After we calculated the difference in angle we easily calculate the new angle for the entire model. During the rotation of the model to the LED, we can get problems with the two point solution, we circumvent this by investigating the sign of the x and y coordinates, if both are negative we know that π should be added to the solution. We don't investigate the case x is negative and y positive because we don't expect that kind of tilt that would result in this case. Figure 3.9 shows the interesting cases with respect to rotation.

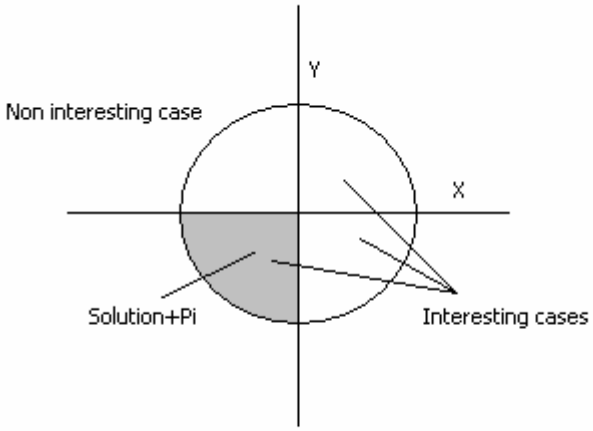

Figure 3.9 – showing the cases that are interesting for the tilting of the face. This is used to pick the right solution after using sine and cosine.

The distance between the LED and the distance between the UL and UR is used to calculate the scaling factor and then scale the model to fit. The MatLab code rotator.m shows the implementation of this.

3.8 Deformation of the shape model.

The update of the model is done using the formula described in (7) and (8). The formula uses the difference between the mean shape and the shape that is to be fitted. We know the mean shape from our annotations, the unknown shape is found by using same technique as when allowing for movement in the LED. That is; we investigate a small box around each landmark and calculates the median x and y position in that box. This new median estimate is used as a landmark estimate of the unknown shape. Calculating the difference and adding the dispersion in (8) we get the parameter estimates and can easily deform the model afterwards. If the box around the landmark is empty the best estimate of the landmark is the previously known landmark which is then used.

The update has been tried with two different approaches: 1) letting the updated model be used as the basis for the next update. 2) Letting the mean shape be used as the basis of the update at each iteration, i.e. letting the model be reset at the beginning of every iteration. It is clear that 1) seems to be the most dynamic approach but on the other hand this enables the model to deform into shapes that are not really faces but yet inside the region of deformation, e.g. if one side is being pulled away, there is nothing stopping it only delaying it due to the constraints in movement per iteration, this is not a problem with 2). Both approaches are shown below in figure 3.10 and 3.11

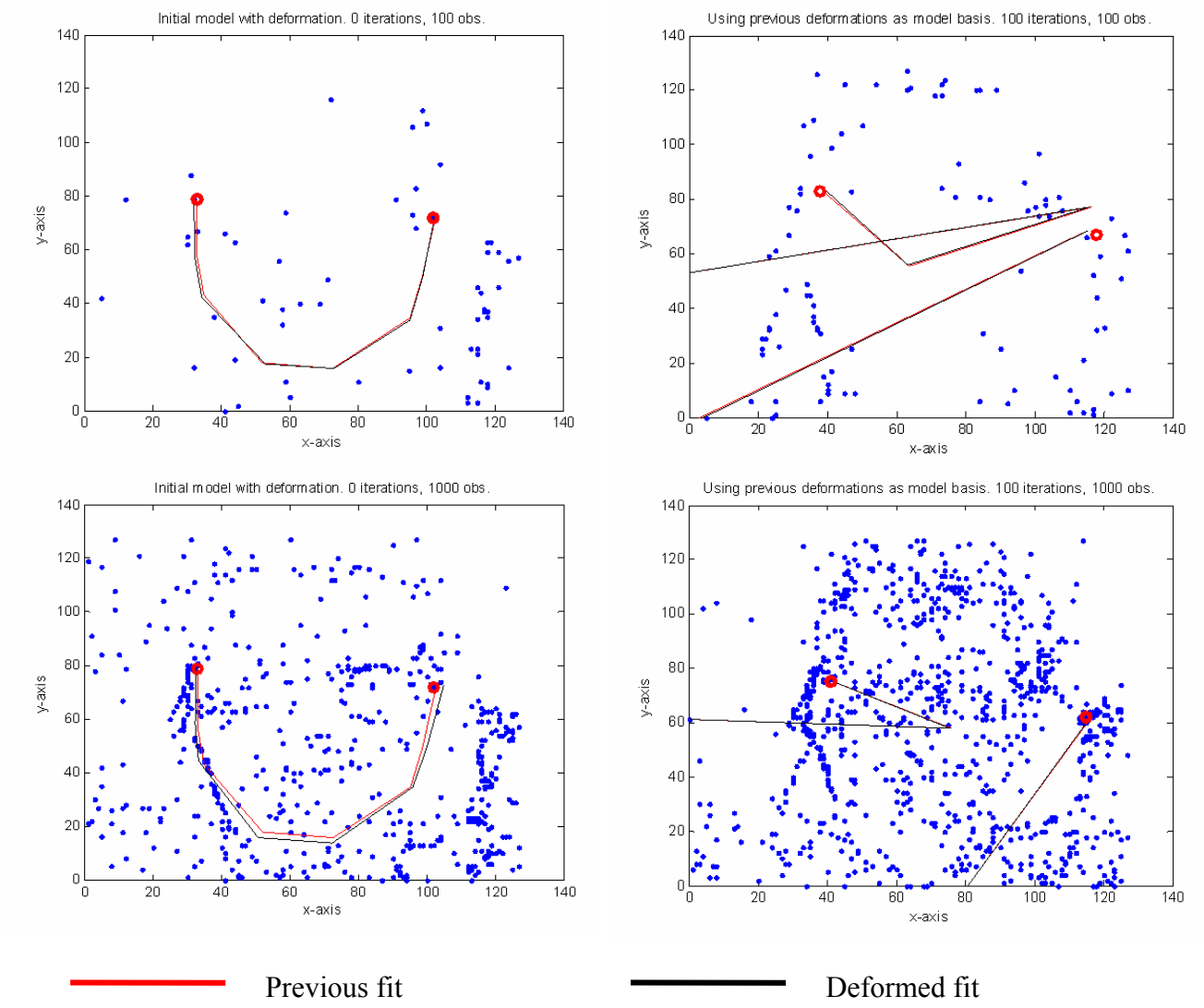

Figure 3.10 –top Left is shown the initial estimate of the face with 100 events. Top right is shown the deformation of the model after 100 iterations. The model landmarks have drifted to far apart, even though restrictions on movement has been applied. Bottom left is shown the initial model estimate with 1000 events. Bottom right is shown the deformation of the model after 100 iterations. The black line is the deformation, the red line is previous fit.

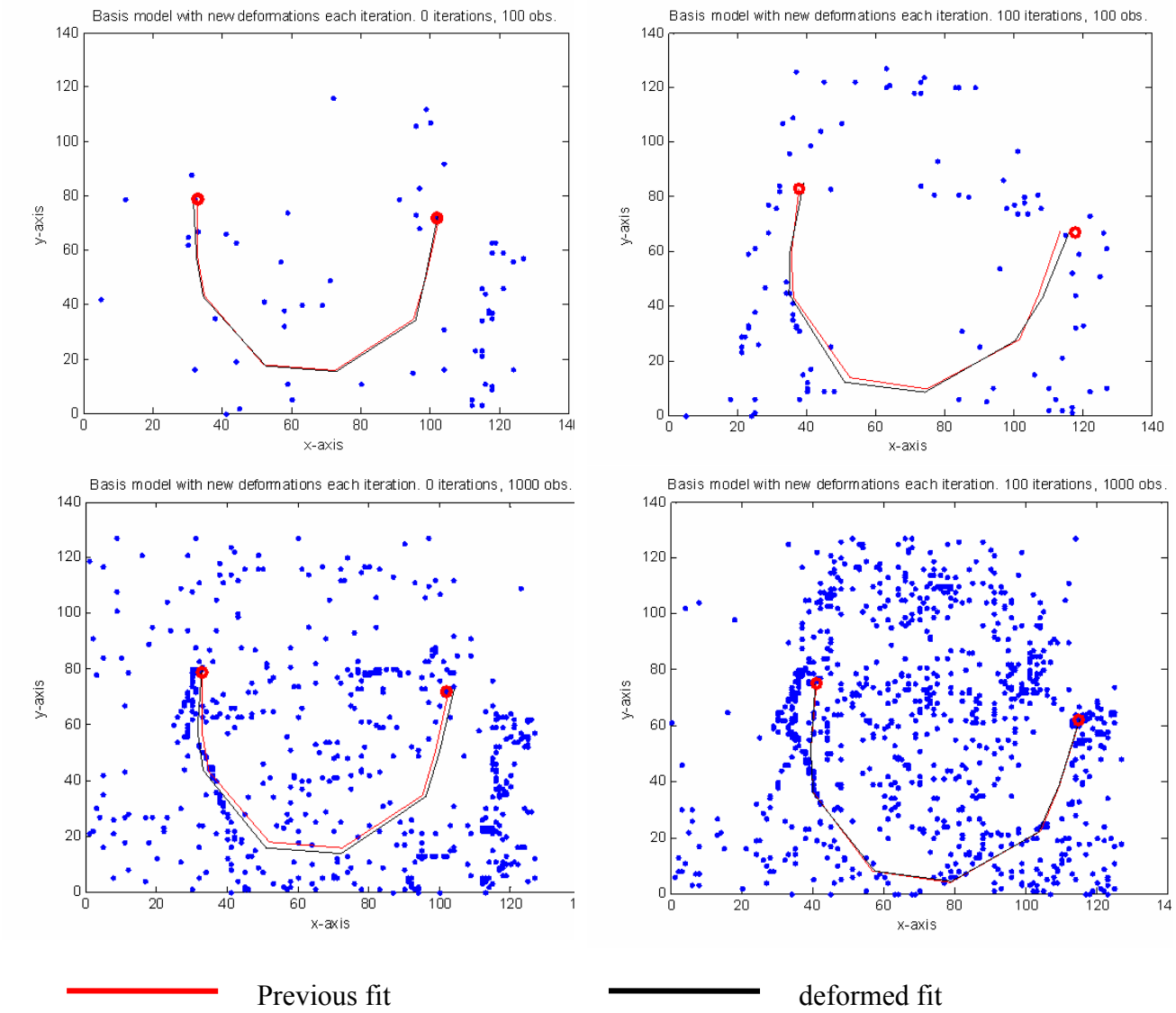

Figure 3.11 – Top left, initial estimate of the model, top right model deformation and movement after 100 iterations. It is seen that the model still mimics a face because of the reuse of the initial estimate at every iteration. Bottom left, initial estimate with 1000 events. Bottom right, model movement after 100 iterations, the model still captures a face shape, though it has been stretched due to some accidental noise, showing the instability of the LED.

In the MatLab script we tried to use overlapping data, e.g. 1-1000, then 500-1500 and so on, not only trying with overlapping 500 events but different sizes. Not one of them made any difference in the result of the deformation of the model, or even influencing the movement of the LED it just required more iteration before the same effect was seen.

4 Modeling the mouth

This section uses an annotated model of a face including the annotation of an average mouth, giving us an initial estimate of the placement of the mouth with respect to the face. The modeling of the mouth to recognize if the mouth is open or closed is divided into 5 parts each concerned with a very distinct feature of the mouth. 1) defines where the mouth is expected to be. 2-3) rotation of data and area from 1, making it easier to find the data that belongs to the mouth. 4) Estimation of the median in each cell. 5) Rotating back to the initial angle and scaling to fit the face.

The mouth is limited in its movement; this restriction is enforced by not letting it jump outside a boundary box, shown in figure 4.1. Though it is possible for the mouth to jump outside the box, at each iteration if there is not enough data to update a specific landmark from the mouth and hence no update is done. This will correct it self when the landmark in question updates it self.

The boundary box is defined by four landmarks. The landmarks are defined by looking at the height and width of the mouth. Finding the upper bound for the box is done by taking the mouth landmark with the largest and smallest y-value and adding respectively subtracting a factor to this. The same is done for the x-axis to give a left and right bound for the box. This gives us with respect to the model an expected area where the mouth can move. The box is linked to the mean shape of the face.

The enclosure is located with an upper bound by the landmark just under the ears and lower bound just below the landmark representing the cheek. This has been chosen since looking at photos and mirrors, and a model of a face and mouth made it seem like a good size of the box with plausible movement restrictions for the mouth. The box is rotated and scaled like the face model, and an initial estimate of a mouth has been modeled inside of this box. Figure 4.1 shows the initial position of the face, box and mouth.

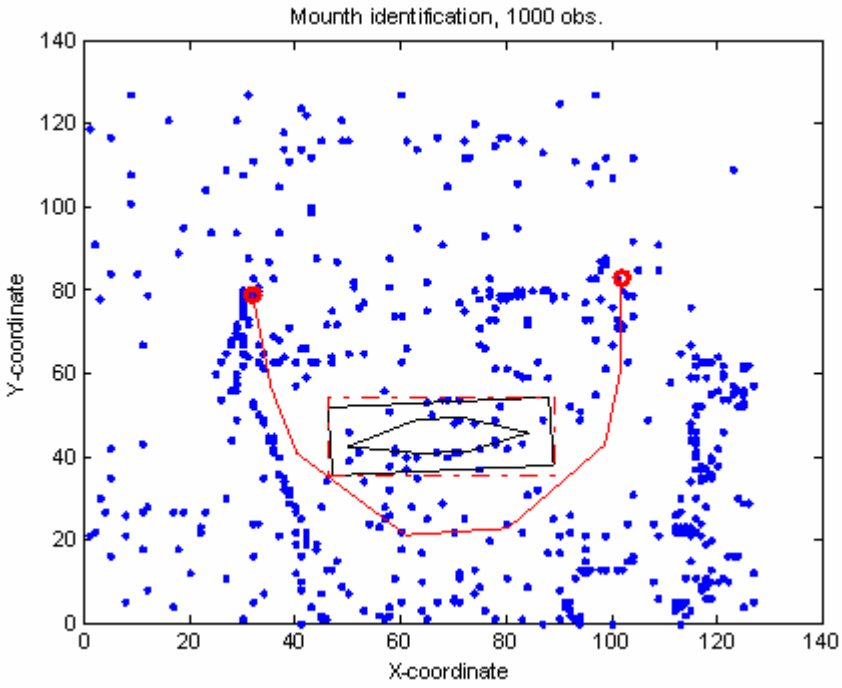

Figure 4.1 – Inserted face model with mouth and boundary box for the mouth, limiting the movements of the mouth. The stippled box

around the mouth is the box used for finding relevant observations.

When trying to model the mouth only observation that lay on or inside the boundary box is to be taken into consideration. Finding the relevant observations that are inside the boundary box could be difficult when the boundary box is tilted. To avoid testing all observations op against a tilted boundary e.g. an equation, the following 2 steps are done. First; defining an outer box around the boundary box, this outer box is the smallest horizontal box that includes all four landmarks from the boundary. Only observations inside the outer box are possibly relevant for the modeling of the mouth (stippled box in figure 4.1). Second; instead of finding an equation on each boundary vertex and check each observation against, to ensure it is inside the boundary, we rotate the information inside the outer box such that the boundary box becomes horizontal and the data aligned to this.

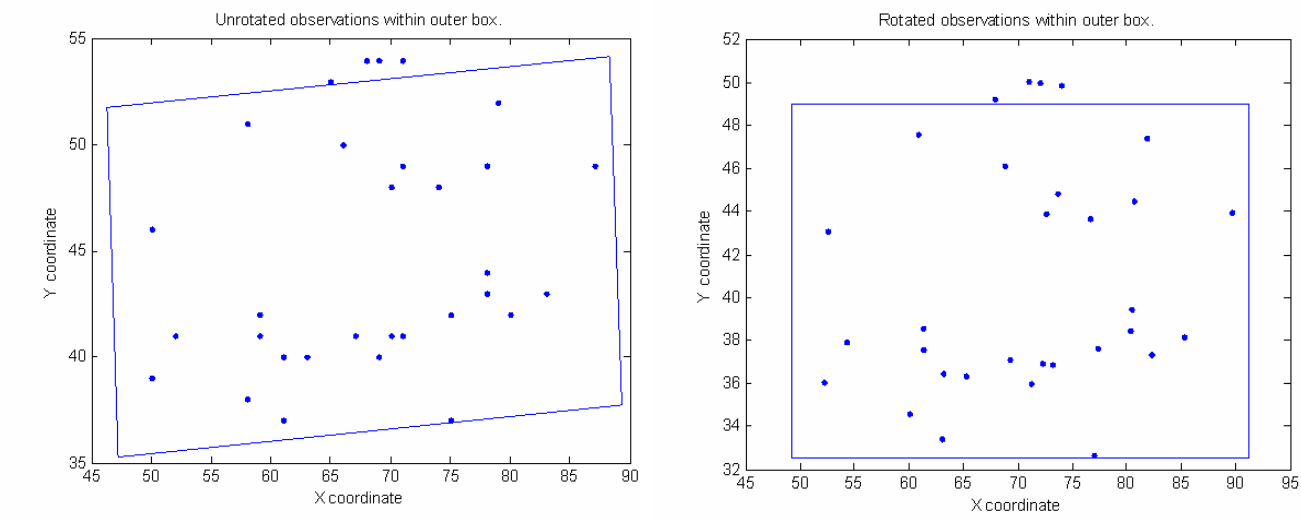

Figure 4.2 – Rotation of the data in the boundary box. Left is shown how the data are collected from the camera. Right, shows the rotation of the data to easily enforce the constraints.

To find the possible relevant observation out of n events, we only accept observations that lay inside the outer box, instead of doing it directly on the boundary box. This is done for simplicity making it a cheap way to find the possible relevant events. Only the observations that are deemed possible relevant (inside outer box) are being rotated. After rotation a logical statement investigates each event to see if it is inside the boundary. This is easily done by knowing the upper, lower, left and right boundaries. Figure 4.2 (left) shows the initial rotated boundary with the outer box, and the possible events. Figure 4.2 (right) shows the boundary after rotation. It is clearly easier to look at the rotated data and see which events are inside the box.

To rotate the box and data we first need to find the angle of rotation, this is easily found by finding the difference in y between the two x-axis coordinates.

$$\text{Slope} = \frac{y_2 - y_1}{x_2 - x_1} \tag{10}$$

This is the same as used to calculate the tilt of the face. The slope makes it possible to calculate the angle of the tilt θ. θ Is then subtracted from all calculated angles of the boundary landmarks and

the observation angles. All rotated observations and landmarks needs to be investigated for correct solution due to the nature of angular-relations having two equal solutions, seen in section 3.7, figure 3.9. Using the upper left landmark as centre of rotation all solutions has to be checked if the are to the left of this point. If they are we also need to know if the observation is above or below the rotational centre.

- If the observation is to the left of the centre of rotation pi is added to the angle to ensure that the observation is mapped to the correct side of the centre.

- If the observation is below the centre of rotation the angle is set to negative.

After the tilt the relevant events are easily and cheaply calculated.

4.1 Calculating the median of the mouth.

The modeling of the mouth has is being conducted inside the boundary box. The box is divided into 6 areas as shown in figure 4.3.

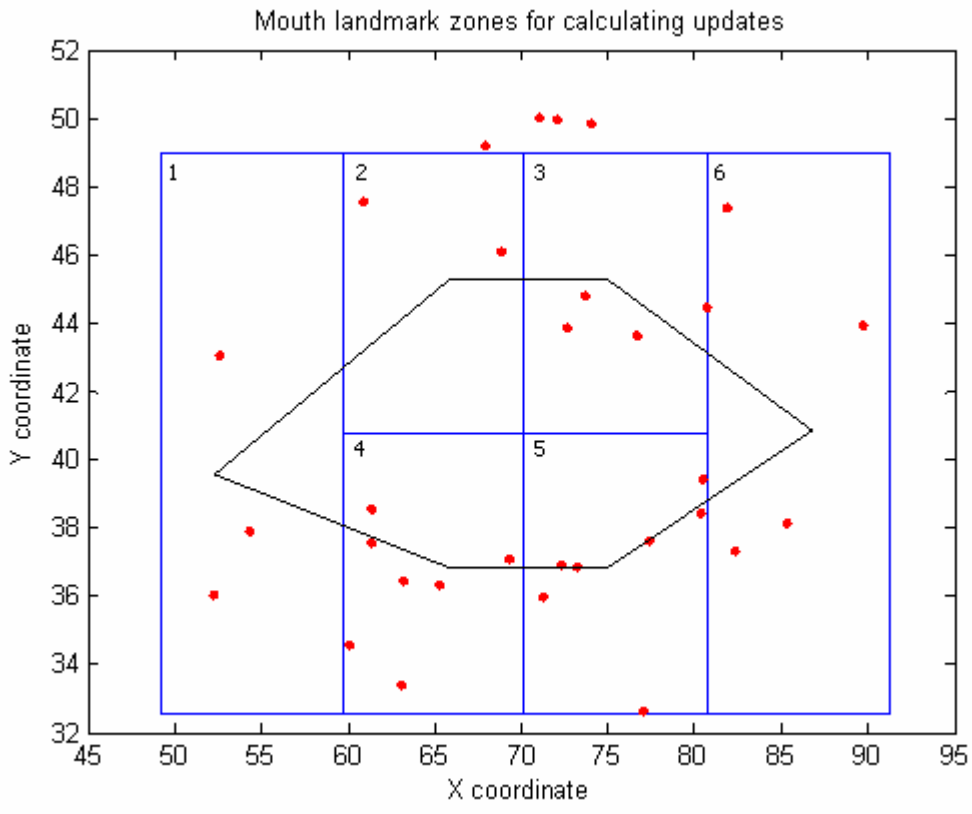

Figure 4.3 – The boundary box has been divided into smaller zones, where individual updates for the mouth landmarks are done. The points represent observations.

From Figure 4.3 we see how the boundary is divided into smaller zones. These zones are used to calculate the best estimate of the mouth landmark. A landmark cannot leave its zone. The zones have been divided into 6 areas with the left and right area being biggest. This has been done in order

to give each landmark an own zone, where it can be calculated independently from the other. The landmark is calculated as the median of x and y. Some restrictions have been imposed on the landmarks;

- The two landmarks composing the upper lip (2+3) are calculated independently, but an average is calculated between them in order to restrict the mouth of getting an unnatural shape. This is likewise done with the lower lips.

- Zone 1 is actually divided into an upper and lower zone, since there is only one landmark on the left edge of the mouth the median of the lower and upper zone is calculated and then the mean of the upper and lower is calculated and used as an estimate of the landmark.

Interchange of median and mean is done because the median gives us a good estimate that is robust towards outliers, but when we try to find the median of two points this is just the same as taking the mean between them. Observations lying directly on the line between two zones are always regarded as belonging to the left and/or lower zone.

It was considered to let the user define the minimum threshold number of observations in each zone, only updating the model if there was enough data in the zone to supersede this threshold. Often there will not be enough data in each zone to provide just 5 observations, so this was not implemented. The main reason was that the danger of the mouth lying outside the boundary will increase significantly, and thus distort the image of the mouth; maybe even put the mouth outside the face. Thus only if there is no observations in the zone will the previous estimate will be used.

Finally the mouth was not modeled using ASM, because the poor control of the movement of the landmark due to small samples. Hence the mouth was annotated from the same 14 images as the face model used, but the variance was not included into the model, instead the ad hoc approach described above was used.

4.2 Mapping back to the tilted face

Only the estimated landmarks from the mouth are being mapped back to the original tilt. This is done by calculating from the previous centre of rotation to each landmark and adding the tilt angle that was previously subtracted; we get to the original tilt again.

4.3 In action

The mouth has been implemented in MatLab and the results are shown below in figure 4.4

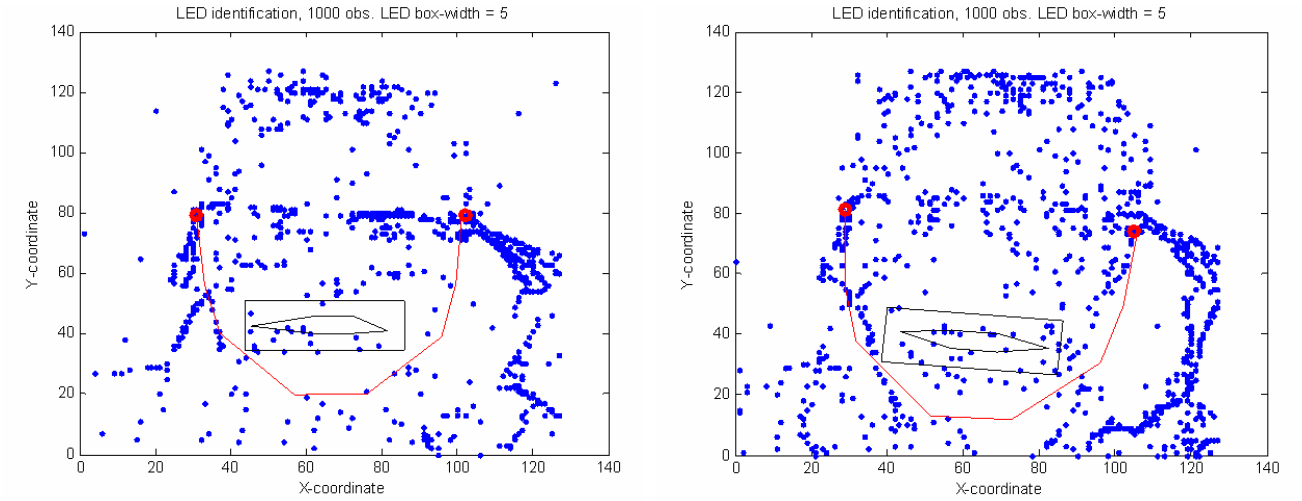

Figure 4.4 – left, the deformation of the mouth when there is no data is done from the previous known estimate. Right, the mouth tilts, scales and deforms. On both pictures is seen the boundary box.

It has been observed that the mouth can jump outside the boundary, this should only happen when there are not enough events to update and an old estimate has to use. Unfortunately also a resetting error has not been located in the MatLab code, making the mouth able to deform unnaturally. This is seen in figure 4.5

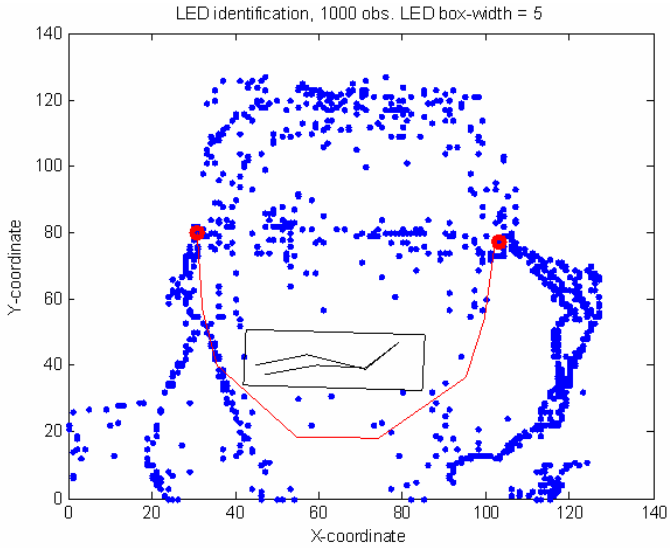

Figure 4.5 – unnatural deformation of mouth, with break up of lips.

It is uncertain if these errors have been eliminated in the java Filter due to slight difference and stricter syntax eliminating small bugs.

Again we have tried to use overlapping data and the result was the same, the only difference is its smoother in it's movements but the result is the same, just using more computations for it.

5 Implementation into JAVA

The project has been build in MatLab and then implemented JAVA as a filter for the jAERViewer. Only the part of the MatLab code that is responsible for dynamic computation e.g. detection of the face has been implemented, while the initial face-model calculations has been copied directly into an array in JAVA, this means that a new face model with e.g. more landmarks has to be calculated in MatLab and then hard coded into the JAVA HorizontalFaceModel file

The files that were implemented into JAVA from MatLab are:

Name	Job
FaceTrack	Initial filter, equivivalent to Face_track.m.
Descriptive	Does descriptive statistics. Incl. Median calculation.
FollowLED	Maintains the LED position, rotation and scaling.
sort	Sorts a vector with either SelectionSort or Heap_sort.
Mund	
HorizontalFaceModel	Contains the model

Table 5.1 – Overview of the different classes in the filter implementation.

The following will only describe the major differences between the MatLab code and the JAVA implementation. The implementation in MatLab has focused on getting the techniques to work, while the JAVA implementation has focused on efficient implementation, e.g. avoiding unnecessary loops. Also a more strict approach to variable declaration, using short, and float where ever possible instead of integers and double. This has an impact on resources used, refer to table 5.2

Type	Storage	Min value	Max value
short	16-bit	-32.768	32.767
integers	32-bit	-2.147.483.648	2.147.483.647
float	32-bit	7 significant digits	
double	64-bit	15 significant digits.	

Table 5.2 – Resource difference between the variable types in JAVA. [6].

From table 5.2 it is clearly seen that the amount of resources used can be substantial reduced, using short and float. It is possible in this implementation to use short and float, as short is used for the pixel position, and float for the face and mouth landmarks, that doesn't need more than 7 digits of precision.

The techniques often runs through the complete vector of observations, this was easily done in MatLab due to native commands that each implement efficient methods for e.g. the median. Implementing into JAVA focused on running through the vector as few times as possible, and each time do as many computation and logical statements as possible to avoid using more loops than necessary.

The script for deforming the face shape has not been implemented due to the final effect on the face shape; refer to "deformation of the shape model", these results made it not worth while to implement this feature.

5.1 Calculation of the median

MatLab uses the command *median* to calculate the median of a vector. JAVA has no equal command; hence this method had to be implemented. Calculating the median is a two step procedure, 1) sorting the vector, by far the more demanding of the two steps, 2) finding the middle element of the vector. The reason for using median is due to its robustness towards extreme values.

To sort a vector or an array we implemented two efficient algorithms; SelectionSort [6] and Heap Sort. SelectionSort contains two nested for-loops, which makes the worst case running time of an n-sized vector equal to $n*n = n^2$ when the vector is completely sorted in reverse order, e.g. increasing instead of decreasing. The best case running time is n for the case when the vector is completely sorted in the required way, then the inner loop is never executed. This sorting algorithm gives a worst case running time of a vector of 1000 elements of 1.000.000 computations, until termination of the algorithm. In order to keep the computations efficient we also implemented the highly effective Heap Sort. This algorithm is comprised of three concepts;

1) Binomial trees and *heaps*; a tree has several nodes, one node is the root of the tree, and the root is the parent to two other nodes that are called the children of the parent. These children can be parents them selves with two children each, and so on. A tree is build from a vector by running through the vector and defining the following [4]:

   Parent(i) = return $\lfloor i/2 \rfloor$

   Left child (i) = return 2i-1

   Left child (i) = return 2i

   Due to Java's indexing starting at zero, the definition of the children has changed a bit. Running through the vector with the above formulas gives a binary tree, in order to make each element the largest parent, we make use of:

2) Max-Heapify, is a technique to ensure that the parent has the largest value of the three. This method takes as input a vector and an index I (from the vector). It checks if the parent (i) is larger than its children, if this is the case nothing changes. If one child is larger the larger child and the parent swaps place and the child becomes the parent and vice versa. Are both children larger than the parent the largest child becomes the parent. After exchange, the previous parent is now a child, the method continuous to check if the new child is a parent of the next set of nodes and the Max-Heapify is run again.

MAX-HEAPIFY(ARRAY,i)**Error! Bookmark not defined.**

```
1  L ← LEFT(i)
2  R ← RIGHT(i)
3  if L ≤ size[array] and array[L] > array[i]
4  then largest ← L
5      else largest ← i
```

```
6  if r ≤ size[array] and array[R] > array[largest]
7     then largest ← R
8  if largest ≠ i
9     then exchange array[i] ↔ array[largest]
10          MAX-HEAPIFY(A,largest)
```

After the execution of Max-Heapify one just needs to remove the root of the tree and run max-heapify for the entire length of the vector.

3) Build-Max-Heap, is used to sort the vector. Using one half for loop to investigate the entire vector placing the largest element in the root of the tree, removing it, and building the tree again. This in turn sorts the entire vector.

```
Build-Max-Heap(array)Error! Bookmark not defined.
1  size(array) ← length(array)
2  for I ← ⌊length([array]/2)⌋ downto 1
3       do MAX-HEAPIFY(ARRAY, i)
```

The Heap Sort algorithm uses one for-loop, which is only executed array.length/2 times. The building of the binomial tree takes at most O(ln(n)) and happens when the last row in the tree is half full, and hence there will be a complete recursion. In the worst case there are n calls to max-heapify which gives an upper bound worst case running time of $O(n \cdot \ln(n))$. In comparison to SelectionSort with 1000 observations in an array, Heap Sort will sort in 1000 * ln(1000) ≈ 6908 computations instead of 1.000.000. Heap Sort gives a dramatic decrease in number of operations required to calculate the median.

Finally we find the median of a sorted vector as one of two cases.

1) If the vector of n elements has an odd number of observations then (n+1)/2 gives the index of the median.

2) If the vector of n elements has an even number of observations is the mean of n/2 + (n+2)/2.

The calculation of the median is used in many different aspects in this project. The sorting methods are implemented in the JAVA class file called Sorting as two different methods taking as argument an array. This method is not bound to the jAER filter and can be used to sort by just calling the method.

5.2 Other classes and methods.

The JAVA implementation contains 5 further classes. These classes and there prime methods will briefly be discussed.

The overall class in this filter is **FaceTrack** this class is an implementation of the MatLab script face_mund, which controls the flow of the filter. In turn this class calls the other classes to get the

calculations. This class is also generating the array with x and y values for events. It includes several small methods, the most important once being:

- **resteFilter();** this resets the filter to the initial parameters, e.g. the face model is reset.
- **Eventpacket <?> filterPacket(…);** (generic) which is the method that generates the x and y data arrays used for computation.
- **processEvents ();** this method is where the computation of the face and mouth is done, calling other classes.

A very important class is **FollowLED,** this class is doing all computations regarding the LED positions and tilt of the face. It uses 3 different methods:

- **movLED();** is calculating the position of each LED and ensures this cannot move beyond as specific number of pixels at each iteration. The method takes four parameters; LED x, y, box_size. This method returns a float array with the new updated estimates of the position of the LED, the precision of the variable has to be float as it includes a distance measure as well.
- **distance();** this method calculates the distance and is intended to be called from movLED. It takes one parameter; LED, and returns a float.
- **rotator();** is concerned with calculating the tilt of the face, from the LED estimates given in movLED. It takes three parameters; LED, FACEdist, meanshape. It returns a float[][] array with the updated landmarks for the face model.

To calculate the mouth the class **Mund** is used. This class only consists of one method which calculates the entire mouth update, rotation and scaling

- **RotateMouth();** is responsible for estimating the mouth. It takes five parameters; x, y, face model, previous estimate of mouth, mouth boundary box. The method returns float[][] array containing the x and y coordinates of the new mouth.

The initial model of the face and the mouth is contained in the class **HorizontalFaceModel**. It consists of two methods:

- InitialFace(); which contains the face and mouth model in a float[][] array, which is returned when called.
- FACEdist(); calculates the size of the face used for scaling to the LED, it takes as argument a float[][][] array intentionally set to [8][2], the size of the face. It returns a float[][] array with the distance between the first and last (8^{th}) position.

To calculate the median of an array the filter uses the Descriptive class. It has one method;

Median(); which calculates the median of an array. It takes as input a float array, and returns a float value. The deformation has not been implemented into java, since it did not make any significant difference to the face.

Below is shown a graphic of the filter and the sequence of calls, the control flow seen in figure 5.1.

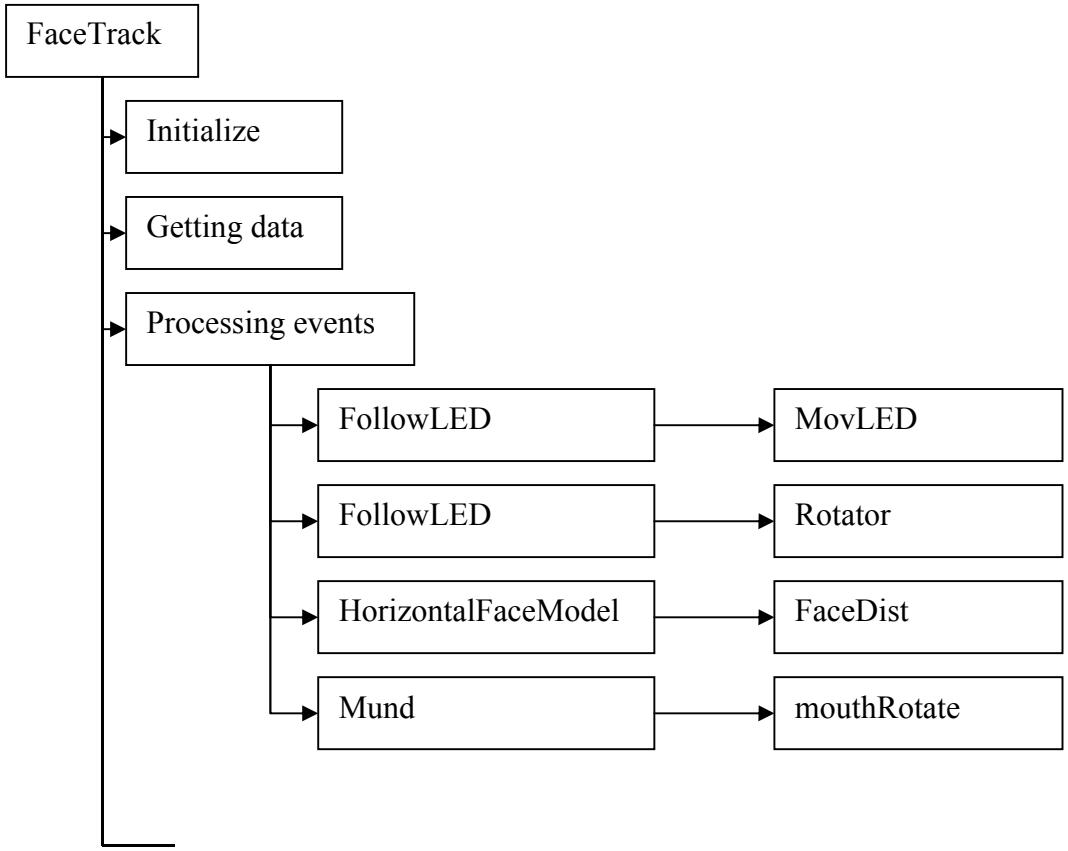

Figure 5.1 – Control flow in the jAER filter doing facetrack.

In figure 5.2 the GUI is seen, this is taken from the general jAER GUI

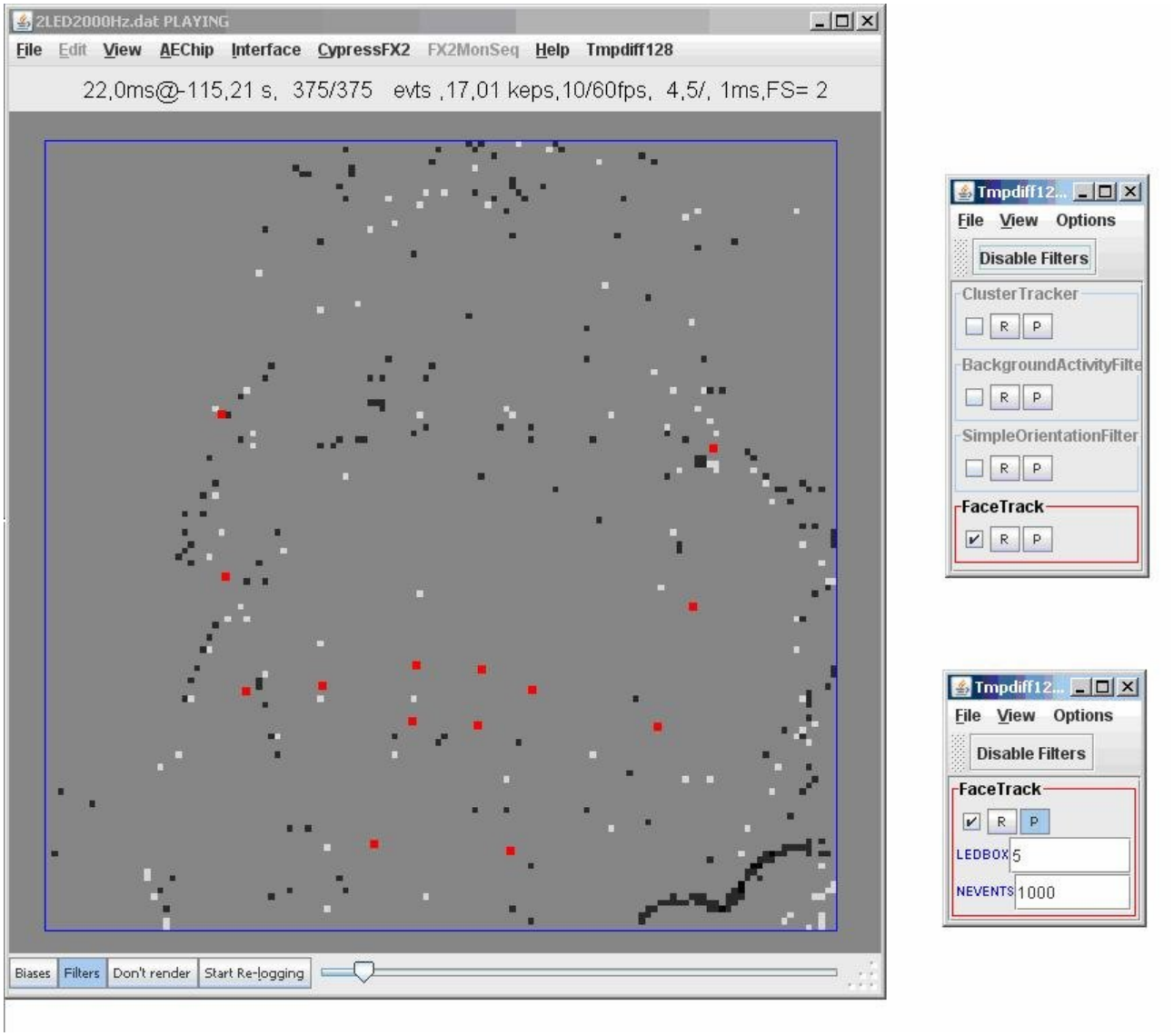

Figure 5.2 – The GUI of the filter, the filter is seen active, the read dots are estimated landmarks of face and mouth. To the right is seen the toolbar with the parameter the user can adjust, number of events before estimation, and size of LED movement per iteration.

6 Computational Effort of filter

Most instructions done on a processing unit is done in discrete time. Initializing variables and evaluating statements are fixed and dependent on the language used and the processor. The filter that has been implemented in this project is no exception, but rather than evaluating every variable declaration and statement we will focus on the computational expensive methods.

The filter is comprised of 6 different classes most of them include different methods, some are even used several times. Below is shown a table with the different classes and what the expected running

time is. If stated “fixed” this means the running time is always the same and will be regarded as a constant.

Class	Running time
HorizontalFaceModel	Fixed
Descriptive	Fixed
FollowLED	---
Mund	---
Sorting	n*ln(n)
FaceTrack	Fixed

Table 6.1 – Running time of the different classes. “Fixed” means that the class is always run the same number of times, and thus always runs a deterministic number of operations.

What is really important is loops that are executed a random number of times, due to some constraints. This is seen for instance in the Sorting class, where a list is sorted, is this list is already sorted the sorting will be even faster than the stated worst case running time. We will look deeper into the two classes from table 6.1 that has unknown running time.

FollowLED has one interesting method, which we will investigate:

- **movLED**, controls the movement of the LED. It takes as input the data array created in FaceTrack that has the length n (defined by user). It has a for-loop running through the entire array once to classify which observations are inside the LED boundary (maximum movement of LED pr iteration). The loop takes time n, afterwards *Sorting* is called 4 times, to calculate two sets (x, y) of medians. Each call to *Sorting* takes n*ln(n) time. Due to the nature of ln the following is valid: ln(n) < 2*ln(n/2). Then the worst case happens when exactly half of the data goes to each of the LED’s where it will be $T = 4 \cdot \left(\frac{n}{2} \cdot \ln\left(\frac{n}{2} \right) \right)$. This method is clearly growing faster than n, from the single loop in the method, and we just state that the running time for movLED is upper bound to:

$$T_{movLED} = 4 \cdot \left(\frac{n}{2} \cdot \ln\left(\frac{n}{2} \right) \right) + O(c_1) \qquad (1)$$

  where O(c) is some constant containing the rest of the methods requirement. The assumption about the data being concentrated around the 2 LEDs is not expected to happen that often, but none the less is possible and when this constellation happens it cannot become slower to calculate. The 2 last methods of FollowLED are done in fixed time.

- Mund only includes one method *rotateMouth*, calculating the entire mouth part of the model. This method shares huge resemblance with the *movLED* and rotator combined. There are a couple of loops in this method but the expensive part is when cell medians are calculated. This operation is done 16 times (2 times for each cell), and with ref to *movLED*, this gives a worst case scenario when all collected data fall inside the mouth region, and specifically so that each cell gets exactly 1/8 of all data. This gives a running time of

$$T_{Mund} = 16 \cdot \left(\frac{n}{8} \cdot \ln\left(\frac{n}{8} \right) \right) + O(c_2) \qquad (2)$$

Where $O(c_2)$ is a constant, including the loops. The filter makes in each iteration one call to each of the methods *movLED* and *rotateMouth* giving a total worst case running time of:

$$T_{total} = T_{Mund} + T_{movLED} = 16 \cdot \left(\frac{n}{8} \cdot \ln\left(\frac{n}{8} \right) \right) + O(c_2) + \; 4 \cdot \left(\frac{n}{2} \cdot \ln\left(\frac{n}{2} \right) \right) + O(c_1)$$

Pulling 4 outside the parenthesis

$$= 4 \cdot \left(4 \cdot \left(\frac{n}{8} \cdot \ln\left(\frac{n}{8} \right) \right) + \left(\frac{n}{2} \cdot \ln\left(\frac{n}{2} \right) \right) \right) + O(C)$$

Multiplying 4 into the first parenthesis

$$= 4 \cdot \left(\left(\frac{n}{2} \cdot \ln\left(\frac{n}{8} \right) \right) + \left(\frac{n}{2} \cdot \ln\left(\frac{n}{2} \right) \right) \right) + O(C)$$

Pulling n/2 outside the inner parenthesis

$$= 4 \cdot \left(\frac{n}{2} \cdot \left(\ln\left(\frac{n}{8} \right) + \ln\left(\frac{n}{2} \right) \right) \right) + O(C)$$

Simplifying using ln(a) + ln(b) = ln(a*b)

$$= 4 \cdot \left(\frac{n}{2} \cdot \ln\left(\frac{n^2}{16} \right) \right) + O(C)$$

Multiplying 4 into the parenthesis

$$T_{total} \qquad = n \cdot \ln\left(\frac{n^2}{16} \right) + O(C) \qquad (3)$$

Which is the final upper bound running time for the worst case scenario, there is one problem though, in this analysis we didn't look into the fact that the data is either used for (1) or (2). Meaning this upper bound will never be reached, but (1) or (2) is likely to be experienced.
This has shown that the filter has a worst case running time when the sorting algorithm is used on the entire collected data. Also it underlines the importance of the implementation of HeapSort as preference above the likewise implemented SelectionSort with running time of n^2 each call, which would have been much more expensive. The filter and the MatLab script was easily run on a standard IBM T41, with Pentium M 1600 MHz, and 1 Gb of ram.

7 Ideas for future work:

This project has just briefly looked into the nature of event based data used in a framework of statistical image analysis. Though this report shows the thing I tried there are still some open questions that I didn't find the answer for:

Inward deformation:

The face has been implemented in MatLab to deform in any direction, this is probably not very reasonable since the LEDs, used for positioning the model, is set on glasses, this would mean the face is a bit smaller than the LED actually dictate and hence the face should be brought to deform inward only. This was very briefly looked into but deemed to time consuming. Doing this would probably give a better fit of the deformation and even allowing the model to be dynamically updated, without drifting as shown in XX6.

More stable LED estimate, possibly using PDF:

I was unable to stabile detect the LED, and thus imposed constraints on how far it could per iteration, this comes from the problem described in the section "identifying the LED". It would probably be possible to estimate the LED better, incorporating the PDF of the LED and thus knowing more about the movement.

Value of each event (+/-)
Looking into the information of the value of each event (+/-) could probably improve the model, but exactly how to weight the value from each event in the data vector was not something I could grasp. And as previously stated, it didn't immediately seem useful in for the model.

Use of inertia between events
Especially for the mouth it would be interesting to see if a better estimate on the movement could be made using inertia of the pixels. This would enable the model to know where the lips are heading, e.g. opening or closing the mouth, and thus leading to a better estimate of the mouth.

8 Conclusion

This project has been difficult because of its outer shell belonging to image analysis, but in fact it has little to do with frame based analysis, even sequential frame based is not equal, this is images on event basis and in a series. There is very little apparent similarity between these fields, thus has much of my work been to grasp the new kind of data this was, and trying to work with them.

It was possible to identify an alternating LED light in an event stream from the camera via medians and through this estimate the position of a face. Though the estimated position of the LED first became stabile when setting constraints on the movement per iteration. Using an initial guess for the LED it was then possible to track two LED and use them as waypoints for the face. This gave us a reasonably good estimate and face track, that was usable to locate the face for/of the user.

Using ASM on the event based data to deform the face was not successfully achieved. When trying to deform the model, it started drifting only bound to the two LEDs. Constraining the deformation to be reset after each iteration over an event stream (e.g. n = 1000) it proved to be stable but with little effect as the data were too scarce distributed over the entire sensor.

Using the LED and the face shape, it was possible to estimate the position of the mouth, using a minimum boundary box (MBB) the mouth could be estimated and in MBB, a cell structure was used to calculate each landmark independently. The mouth deformation was done using only medians in each cell and not AAM due to the lack of success from the face. The result was very successful and the mouth was able to reasonably deform into open and closed, though with some errors. Especially when there is movement in the face the mouth will be deformed by accident due to the nature of the implementation.

Even when using event streams of 1000 events, the data in small areas become very scarce, making statistics very difficult. Finding an optimal length was not investigated and the implementation lets the user define it.

The two models were implemented into JAVA to run approximately real time, using a buffer. The implementation was focused on making the code more efficient that the draft from MatLab, this was achieved through stricter declaration of variables, and fewer loops. To calculate the median of this buffer an implementation of HeapSort was made, making the sorting of large arrays very efficient.

The analysis of the running time of the jAER filter showed that the most expensive part of the filter is the frequent sorting of data, this in terms justifies the implementation of the Heap sort algorithm making the sorting of n number go from n^2 to n*ln(n). The filter and the MatLab scripts run without any problems on a standard pc with a Pentium M 1600 MHz and 1 Gb of ram.

To try the demo supplied with this project, refer to appendix A, and the read me.txt in the COGAIN folder of the supplied code.

9 Acknowledgement, in random order

I would like to thank Tobi Delbruck for his advice and good ideas. For his great patience trying to teach me JAVA, and all the cool tricks, I'm sure some of it will stick, thanks ☺

Also I would like to thank INI for letting me in and making me fell welcome, and for using their equipment, all in all being a very friendly institute.

I would like to thank EU/COGAIN for grants regarding my stay in Zürich

Finally I'd like to thank Bjarne K. Ersbøll for directing my attention towards this project, contacts with EU/COGAIN, and for good ideas and advice and believing in me. And finally for cheerful mails, keeping my head up high when the project seemed to far out of reach, thanks ☺

10 Bibliography:

Statistical shape models/Active appearance models:

[1] T.F Cootes and C.J. taylor; Statistical Models of Appearance for Computer Vision, University of Manchester, March 8, 2004.

[2] Ian L. Dryden and Kanti V. Mardia; Statistical Shape Analysis; Wiley

MatLab programming:

[3] Amos Gilat; MatLab, An Introduction with applications, 2nd ed. Wiley, ISBN 0-471-69420-7

JAVA implementation and algorithms:

[4] T. H. Cormen, et al; Introduction to algorithms, 2nd ed. MIT press, ISBN 0-262-53196-8

[5] Bruce Eckel; Thinking in JAVA, 9th ed. Prentice Hall, ISBN 0-13-027363-5

[6] Lewis et al; Java, software solutions, 3rd ed. Addision-Wesley, ISBN 0-201-78129-8

Mathematics:

[7] E. Kreyszig; Advanced engineering mathematics, 8th ed. Wiley, ISBN 0-471-33328-X

[8] H. Madsen; Time series analysis, Kgs. Lyngby 2001, lecture notes.

11 Appendix A: Script for running the different MatLab codes.

11.1 COGAINdemo.m

```
%This script is intended for demonstration of the COGAIN facetrack project
%done by Alexander Tureczek in the fall of 07 and spring of 08. In each of
%the demos the user can tweak around with the parameters to see the effect.
% The parameters that can be altered are marked with a start under them.

%@Alexander Tureczek
clear all


%In order to run the demo, one of the below datasets has to be uncommented,
%e.g. remove the "%" in front of the name.
disp('Welcome to the deomostration of the Silicon retina face track project,')
disp('In the following 4 different demos will be displayed one by one, after')
disp('each demo you will have the choice to see it again. It has to be stated')
disp('that due to the amount of choices the user has in this demo, errors
might')
disp('be encountered, if so please restart the demo with new inputs.')
disp('Press enter to continue.')
disp(' ')
disp('Alexander Tureczek, june 08')
input(' ')
clc



disp('Choose your test data, options 1-5, Default is 2 kHz')
disp(' ')
disp('1: 300kHz data with 2 led')
disp('2: 500kHz data with 2 led')
disp('3: 1000Hz data with 2 led')
disp('4: 1500Hz data with 2 led')
disp('5: 2000Hz data with 2 led')
disp(' ');
data=input('Please make a choice: ')

if data==1
    [allAddr]=loadaerdat('C:\COGAIN\data\2LED300Hz.dat');
elseif data==2
    [allAddr]=loadaerdat('C:\COGAIN\data\2LED500Hz.dat');
elseif data==3
    [allAddr]=loadaerdat('C:\COGAIN\data\2LED1000Hz.dat');
elseif data==4
    [allAddr]=loadaerdat('C:\COGAIN\data\2LED1500Hz.dat');
else data==5
    [allAddr]=loadaerdat('C:\COGAIN\data\2LED2000Hz.dat');
end

[x,y,pol]=extractRetina128EventsFromAddr(allAddr);

%face_mund is the annotated model that has been used throughout the
%project.
load 'C:\COGAIN\Model\face_mund';
```

```
%Sets the pause between the iteractions in seconds.
disp(' ');
disp('Choose the pause between model update in seconds. recommended is 0.0001,
')
ppause=input('to get a smooth demonstration: ');
disp(' ')
disp('Choose the size of the LED movement box (value set in pixel), 5 is')
LEDsize=input('recommended. WARNING: A TOO HIGH VALUE MIGHT CHRASH THE PROGR: ')
clc

R=1;
while R==1
    disp('%--------------------------------------------------------------------')
    disp('%                                                                    ')
    disp('%  Demo 1: tracking the face and mouth, no deformation of face     ')
    disp('%                                                                    ')
    disp('%--------------------------------------------------------------------')

    face_mund(x, y, 1, 1000, faces, LEDsize, 3, ppause)
    %               *   *           *  *
    R=input('press 1 to see the demo again, press something else to continuoue
the demo')
end
clc

%Loading new model.
load 'C:\COGAIN\Model\omrids';
R=1;
while R==1
    disp('%--------------------------------------------------------------------')
    disp('%                                                                    ')
    disp('%  Demo 2: tracking the face, deformation of face, resetting the   ')
    disp('%  face at each iteration.                                          ')
    disp('%                                                                    ')
    disp('%--------------------------------------------------------------------')

    figure

    face_mund_deform(x, y, 1, 1000, faces, LEDsize, 3, ppause)
    %                       *    *          *  *
    R=input('press 1 to see the demo again, press something else to continuoue
the demo')
end
clc

R=1;
while R==1
    disp('%--------------------------------------------------------------------')
    disp('%                                                                    ')
    disp('%  Demo 3: tracking the face, deformation of face, using estimate  ')
    disp('%  for next iteration.                                              ')
    disp('%                                                                    ')
    disp('%--------------------------------------------------------------------')

    figure
    face_mund_deform_dynamic(x, y, 1, 1000, faces, LEDsize, 3, ppause)
    %                             *    *          *  *
```

```
    R=input('press 1 to see the demo again, press something else to continuoue
the demo')
end
clc


disp('%--------------------------------------------------------------------')
disp('%                                                                    ')
disp('%  Demo 4: tracking the face, deformation of face, using estimate   ')
disp('%  for next iteration, with overlap of 500 events                    ')
disp('%                                                                    ')
disp('%--------------------------------------------------------------------')

figure
face_mund_deform_dynamic2(x, y, 1, 1000, faces, LEDsize, 3, ppause)
%                                 *   *           *   *
disp('End of DEMO')
```

1

12 Appendix B: Important MatLab code

12.1 face_mund.m

```
% Facetracking function.
%Load omrids (faces)
%Load data from retina. (x,y)
%using retinaload.m
%specify the interval length; j, k.

function [hori_meanS]=face_mund(x, y, j, k, faces, box_size, box_size2, min_obs)

%initializing the meanshape, and rotating the meanshape
[meanshape, loadings]=meancent(faces);
[hori_meanS, FACEdist]=rotation(meanshape);

%Defines the border landmarks for the faces excluding the mouth.
face_end=8;

%Width of the face, between the left and right ear.
FACEdist=sqrt(real(hori_meanS(face_end)-
hori_meanS(1))^2+imag(hori_meanS(face_end)-hori_meanS(1))^2);

%used to calculate the increments in the
dif=k-j;

%finding out how many iterations to do.
[m, n]=size(x);

%Not a hole number, thus we end up missing something in the end!?!?!
iter=floor(m/(k-j));

%initial LED identification. This belongs inside the for-loop IF the
%approximation script movLED is not to be used!!!
%[LED1, LED2, LEDist]=LED2id(x,y,j,k)

 LED1=[35,80];
 LED2=[105,80];
 LEDist=(LED2(2)-LED1(2));

%adding the boundary of the mouth to the model.
m=hori_meanS(9:end);
x_add=min(real(m))-0.12;
y_add=min(imag(m))+0.42;
plist1=[complex((min(real(m))-x_add),(min(imag(m))-y_add))];
plist2=[complex((max(real(m))+x_add),(min(imag(m))-y_add))];
plist3=[complex((max(real(m))+x_add),(max(imag(m))+y_add))];
plist4=[complex((min(real(m))-x_add),(max(imag(m))+y_add))];
box_om_mund=[plist1;plist2;plist3;plist4;plist1];
hori_meanS = [hori_meanS;box_om_mund];%
new_mouth=hori_meanS(face_end+1:end-5);


for i=1:iter
    %identifying the position of the LED
    %---- Here is the place where LED2id should be copied in IF LEDmov is
```

```
    %canceled!! ----

    %plotting of the previous LED estimates. Colour used is black
%     plot(LED1(1),LED1(2),'ok' ,'linewidth', 3);
%     hold on
%     plot(LED2(1),LED2(2),'ok' ,'linewidth', 3);

    %Estimation of the movement of the LED since previous iteration.
    [LED1, LED2, LEDist]=movLED(LED1,LED2, x(j:k), y(j:k), box_size);

    %Estimating the face size and position.
    [hori_meanS, FACEdist]=rotator(LED1, LED2, LEDist, FACEdist, hori_meanS);

    %Estimating the updated model from data.
    %[m_update]=mupdate(LED1, LED2, hori_meanS, x(j:k), y(j:k), loadings(:,1:3),
box_size2);

    %Investigating the mouth
    [new_mouth]=mund_rotering(x(j:k),y(j:k),hori_meanS,new_mouth);


    %plotting the data interval
    plot(x(j:k),y(j:k),'.');
    hold on
    %plotting the model on top of the data.
    plot(hori_meanS(1:face_end),'-r');
    plot(hori_meanS(end-4:end),'-k');
    %[mouth_up, hori_meanS(face_end+1:end)];


    plot(new_mouth,'-k')
    %plot(m_update,'-k');
    axis([0,140,0,140]);
    %plot(hori_meanS(face_end+1:end-5),'-r');

    plot(LED1(1),LED1(2),'or' ,'linewidth', 3);
    plot(LED2(1),LED2(2),'or' ,'linewidth', 3);

    [LED1, LED2];
    title('LED identification, 1000 obs. LED box-width = 5')
    xlabel('X-coordinate')
    ylabel('Y-coordinate')

    %updating the model.
    TEST=hori_meanS;
    hori_meanS=complex(real(hori_meanS)-LED1(1),imag(hori_meanS)-LED1(2));
    j=j+dif+1;
    k=k+dif+1;
    hold off
    %hist(x(j:k));
    pause(0.00001);

    %Uncoment if face deformation is done through mupdate.
    %hori_meanS=m_update;
    %hori_meanS=complex(real(hori_meanS)-LED1(1),imag(hori_meanS)-LED1(2));
end
```

12.2 meancent.m

```
%mean centering of shapes
%Using procrustes analysis to calculate the average shape after centering
%of shapes.
%
%mc = mean centering! not standardizing
function [meanshape, loadings, allign]=meancent(faces)

[m,n]=size(faces);

for i=1:n
    mc(:,i)=(faces(:,i)-mean(faces(:,i)))/std(faces(:,i));
end

%formula 3.9 p.44 [1]
z2=conj(mc(:,:))/norm(mc(:,:));
z=mc(:,:)/norm(mc(:,:));
zz=z*z2';

[vec, val]=eigs(zz);

% %Mean shape plot
% plot(vec(:,1),'r', 'linewidth', 5 )
meanshape=vec(:,1);

%alligning all shapes to meanshape.
for j=1:n
    meanallign(:,j)=(conj(mc(:,j))'*meanshape*mc(:,j))/(conj(mc(:,j))'*mc(:,j));
end

allign=meanallign;

%Converting back to real numbers, to calculate dispersion matrix.
%Alligned shapes
xa=real(meanallign);
ya=imag(meanallign);
areal=[xa;ya];

%Meanshape
xm=real(meanshape);
ym=imag(meanshape);
mreal=[xm; ym];
[a,b]=size(mreal);

%calculating the difference from mean to alligned!
for i=1:n
    aresm(:,i)=(areal(:,i))-mreal;
end

%dispersion matrix
disp=(1/(n-1))*aresm*aresm';

%principal component analysis on the dispersion
[loadings, variance, explained]=pcacov(disp);

for j=1:n
pct(j)=sum(explained(1:j));
```

```
end

%-------------------------------------------------------------------------
%-------------------------------------------------------------------------
%
%                  Plotting of Principal components
%
%-------------------------------------------------------------------------
%-------------------------------------------------------------------------

% %Plotting of the variance explanation.
% figure
% plot(pct)
% hold on
% bar(explained)

% %Subplotting with the effect of the first 3 eigenvectors on the meanshape.
% %subplot(height, width, number)
% figure
%
% %first row
% [q,w]=size(mreal);
%
% subplot(3,3,1),
% mode1=[mreal+3*sqrt(variance(1))*loadings(:,1),mreal-
3*sqrt(variance(1))*loadings(:,1)];
% mode2=[mreal+3*sqrt(variance(2))*loadings(:,2),mreal-
3*sqrt(variance(2))*loadings(:,2)];
% mode3=[mreal+3*sqrt(variance(3))*loadings(:,3),mreal-
3*sqrt(variance(3))*loadings(:,3)];
%
% %First row of plots
% plot(mode1(1:(q/2),2),mode1((q/2+1):end,2))
% title('-3 std.')
% ylabel('1. eigenvector')
% axis([-1,1,-0.5,0.5])
% subplot(3,3,2), plot(meanshape)
% title('Meanshape')
% axis([-1,1,-0.5,0.5])
% subplot(3,3,3),
% plot(mode1((1:(q/2)),1),mode1(((q/2+1):end),1))
% title('+3 std')
% axis([-1,1,-0.5,0.5])
%
% %second  row
% subplot(3,3,4), plot(mode2(1:(q/2),2),mode2((q/2+1):end,2))
% ylabel('2. eigenvector')
% axis([-1,1,-0.5,0.5])
% subplot(3,3,5), plot(meanshape)
% axis([-1,1,-0.5,0.5])
% subplot(3,3,6), plot(mode2((1:(q/2)),1),mode2(((q/2+1):end),1))
% axis([-1,1,-0.5,0.5])
%
% %Third row
% subplot(3,3,7), plot(mode3((1:(q/2)),2),mode3(((q/2+1):end),2))
% ylabel('3. eigenvector')
% axis([-1,1,-0.5,0.5])
% subplot(3,3,8), plot(meanshape)
% axis([-1,1,-0.5,0.5])
```

```
% subplot(3,3,9), plot(mode3((1:(q/2)),1),mode3(((q/2+1):end),1))
% axis([-1,1,-0.5,0.5])

meanshape=meanshape;

% [1] Statistical shape analysis, Ian L. Dryden et al. (course note)
```

12.3 Rotation.m

```
%This function rotates the meanshape to an angle where the face is in
%vertical (normal position). Uses polar coordinates to rotate.

function [hori_meanS, FACEdist, r]=rotation(meanshape)

%Translation to center around upper left landmark
hori=meanshape-meanshape(1);

%Defines the border landmarks for the faces excluding the mouth.
face_end=8;

%vector of angles
ang=angle(hori);

%Calculating the radian between the upper left landmark and upper right
%landmark (principal angle = difference between upper left and right
%landmark)
ang=[0;ang(2:end)-ang(face_end)];

new_ang=2*pi+ang;

%calculating the argument of the points
r=sqrt(real(hori).^2+imag(hori).^2);

%Creating the complex xy notation from the polar
new_real=r.*cos(new_ang);
new_imag=r.*sin(new_ang);

%Horizontal meanshape
hori_meanS=complex(new_real,new_imag);

%calculating the size of the model, used for scaling with the true data from
retina;
 FACEdist=real(hori_meanS(face_end));
```

12.4 LED2id.m (NOT USED, but works)

```
%Function for estimating the position of the LED

%function [LED1, LED2, LEDist]=LED2id(x,y,j,k)

%-------------------------
%-------- X-axis --------
%-------------------------
x=x(j:k)
x_new=sort(x);

%Identifying the position of the left median and the right median
%Left median
medL=length(x_new)/4
%Right median
medR=3*length(x_new)/4

%Value of left median in x coordinate
medLx=x_new(medL);
%Value of right median in y coordinate
medRx=x_new(medR);

%-------------------------
%-------- Y-axis --------
%-------------------------
y=y(j:k);

mLy=find(x==medLx);
mRy=find(x==medRx);


medLy=median(y(mLy));
medRy=median(y(mRy));

LED1=[medLx medLy];
LED2=[medRx medRy];

LEDist=sqrt((medRx-medLx)^2+(medRy-medLy)^2);
```

12.5 movLED.m

```
%Method for restraining the movement of the LED for each Iteration.

function [LED1u, LED2u, LEDist]=movLED(LED1,LED2, x, y, box_size)

%Find all the maximum allowed movement for x1.
LED1x=(x>=LED1(1)-box_size & x<=LED1(1)+box_size);

%Estimate the x1 position of LED1
est_LED1x=median(x(LED1x));

%finding the maximum allowed movement for y1 for LED1
y_int1=(x>=LED1(1)-box_size & x<=LED1(1)+box_size); %dobbelt konfekt!!!
est_y1=(y(y_int1));
%Finding all y's in the allowed interval
LED1y=(est_y1>=LED1(2)-box_size & est_y1<=LED1(2)+box_size);

%finding the median of the y1
est_LED1y=median(est_y1(LED1y));

LED1u=round([est_LED1x, est_LED1y]);

%------------------------------
% Doing the same for the LED2

%Find all the maximum allowed movement for x1.
LED2x=(x>=LED2(1)-box_size & x<=LED2(1)+box_size);

%Estimate the x1 position of LED1
est_LED2x=median(x(LED2x));

%finding the maximum allowed movement for y1 for LED1
y_int2=(x>=LED2(1)-box_size & x<=LED2(1)+box_size);
est_y2=(y(y_int2));

LED2y=(est_y2>=LED2(2)-box_size & est_y2<=LED1(2)+box_size);

%finding the median of the y1
est_LED2y=median(est_y2(LED2y));

LED2u=round([est_LED2x, est_LED2y]);

%-----------------------------------------
%--------- Distance between LED ------------
%-----------------------------------------
LEDist=sqrt((LED2(1)-LED1(1))^2+(LED2(2)-LED1(2))^2);
```

12.6 Rotator.m

```
%function rotating the scaled FACE to the LEDs angle and translating into
%the face.

function [hori_meanS, FACEdist]=rotator(LED1, LED2, LEDist, FACEdist,
hori_meanS)

%--------------------------------------------------------------------------
%                           Angle Calculation
%--------------------------------------------------------------------------

%Calculating the slope between the LEDs on the glasses. and calculating the
%angle of the %face in radians. (deegree=(radians*180)/2*pi)
LED_slope=atan((LED2(2)-LED1(2))/(LED2(1)-LED1(1)));

%Defines the border landmarks for the faces excluding the mouth.
face_end=8;

%Model slope known from the previous fit. Using only the upper right and
%left landmarks to determine the angle of the face.
Model_slope=atan((imag(hori_meanS(face_end))-
imag(hori_meanS(1)))./(real(hori_meanS(face_end))-real(hori_meanS(1))));

%Calculating the difference between the model slope and the LED slope, to
%correct for movement since last update of model.
dif_slope=LED_slope-Model_slope;

%Angle of the landmarks of the face model
mod_ang=atan((imag(hori_meanS(2:end))-
imag(hori_meanS(1)))./(real(hori_meanS(2:end))-real(hori_meanS(1))));

%Check which solution to arctan is to be used, atan or atan+pi. The
%counting starts in 2 because hori_meanS has a zero element in (1) and is
%0ne element larger than mod_ang.
for i=2:length(hori_meanS);
    if real(hori_meanS(i))<0 && imag(hori_meanS(i))<0
            mod_ang(i-1)=mod_ang(i-1)+pi;
    end
end

%Calculating the new angle of the model by adding the difference between
%previous model and the new LED angle. And setting position (1)=0
new_ang=[0;(mod_ang+dif_slope)];

%calculating the argument of the points (landmarks)
r=sqrt(real(hori_meanS).^2+imag(hori_meanS).^2);

%Creating the complex xy notation from the polar
new_real=r.*cos(new_ang);
new_imag=r.*sin(new_ang);

%tilted meanshape
tilt_meanS=complex(new_real,new_imag);

%--------------------------------------------------------------------------
%after having found the angle of the face we calculate the scaling of the
```

```
%face.
%-------------------------------------------------------------------------

%Calculating the scaling factor.
s_fact=LEDist/FACEdist;

%Scaling the model to fit the 2 LEDs.
estimate=s_fact.*tilt_meanS;

%distance in real numbers
FACEdist=sqrt(real(estimate(face_end)-estimate(1))^2+imag(estimate(face_end)-
estimate(1))^2);

%the final tilted face model.
hori_meanS=complex(real(estimate)+LED1(1),imag(estimate)+LED1(2));
```

12.7 mupdate.m

```
%Calculating better landmark positions from given data. Using the known
%landmark as an initial guess, and calculating new median for x and y.

function [m_update]=mupdate(LED1, LED2, hori_meanS, x, y, loadings, box_size2)

[n,m]=size(hori_meanS);

x_coor=real(hori_meanS);
y_coor=imag(hori_meanS);

%Setting the first and final position equal to the LED position
m_update(1)=complex(LED1(1,1), LED1(1,2));
m_update(n)=complex(LED2(1,1), LED2(1,2));

%Loop investigating every landmark for better position!
for i=2:n-1

    %interval of interest defined from hori_meanS
    x_int=(x>=x_coor(i)-box_size2 & x<=x_coor(i)+box_size2);

    %Securing that the x and y intervals in question are not empty
    if sum(x_int)==0
        est_x=x_coor(i);
    else
        %X coordinate of the landmark
        est_x=median(x(x_int));
    end

    %Identifying the Y interval and coordinate. First we find the relevant
    %Y's i.e. the y in the interval found above.
    y_relevant=y(x_int);

    %Investigating the Y's in the interval around the modellandmark. If
    %none it is set to the input value already known.
    y_int=(y_relevant>=y_coor(i)-box_size2 & y_relevant<=y_coor(i)+box_size2) ;

    if sum(y_int)==0
        est_y=y_coor(i);
    else
        %Y coordinate of the landmark
        est_y=median(y_relevant(y_int));
    end

    %Delivering the estimates in the same format at received in hori_meanS
    %(complex).
    m_update(i)=complex(est_x, est_y);

end

m_update

%The deformation via loadings can be removed by deleting the below listet
%code. The only m_update from above needs to be returned.

%Calculating the new model fit from the formula 4.4 and 4.5 from T.F. Cootes
```

```
%First finding the difference between the initial estimate and the updated.
x_diff=(real(m_update')-real(hori_meanS));
y_diff=(-1*imag(m_update')-imag(hori_meanS));
%mod_diff=complex(x_diff,y_diff);
mod_diff=[x_diff;y_diff];

%Inverting the loadings vector
t_load=transpose(loadings);

%calculating the model parameters.  (4.5)
para=t_load*mod_diff;

meanshape=[real(hori_meanS);imag(hori_meanS)];

%Calculating the final updated model
m_update=meanshape+loadings*para;

 m_update=(complex(m_update(1:n),m_update((n+1):(2*n))))
```

12.8 mund_rotering.m

```
%functionen mangler stadig at blive at implementere et minimum for hvor
%mange observationer som skal til før en opdatering laves.

function [new_mouth]=mund_rotering(x,y,hori_meanS,mouth)

%Centering point 4, upper left, to 0,0
box=hori_meanS(end-4:end);
%box=box-box(4);

%Calculating the slope between point 3 and 4, to finde the rotaton of the box
box_slope=(imag(box(3))-imag(box(4)))/(real(box(3))-real(box(4)));
%%box_slope=acos()

%-------------------------------------------------------------------------
%
%               rotation of the box, to become horisontal
%
%-------------------------------------------------------------------------

    %finding the slope of every point.
    r=sqrt(real(box).^2+imag(box).^2);
    %the following is done to prevent divide by zero error.
%     r(4)=1;

    %Angle of point
    ang=acos(real(box)./r);
    %Setting the true value r and ang(4) to zero
%     r(4)=0;
%     ang(4)=0;

    %Investigating the solution to chose from the angle calculation.
% %     for i=1:length(ang)
% %         if imag(box(i))<0
% %             ang(i)=-1*ang(i);
% %         end
% %         if real(box(i))<0
% %             ang(i)=ang(i)+pi;
% %         end
% %     end

    %Subracting the difference between the angle and the slope
    new_ang=ang-box_slope;

    new_x=r.*cos(new_ang);
    new_y=r.*sin(new_ang);

    new_box=complex(new_x,new_y);

%-------------------------------------------------------------------------
%
%               rotation of the data, to become horisontal
%
%-------------------------------------------------------------------------

    %finding the relevant data
    %Finde data i og lige omkring kassen omkrng munden.
```

```
    count=1;
    mbox=hori_meanS(end-4:end);
    mx=0;
    for i=1:1000
        if x(i)>=min(real(mbox)) && x(i)<=max(real(mbox))
            if y(i)>=min(imag(mbox)) && y(i)<=max(imag(mbox))
                mx(count)=x(i);
                my(count)=y(i);
                count=count+1;
            end
        end
    end

    if length(mx)>1
    %     %Centering the data to the point box(4)=0,0
    %     mx=mx-real(box(4));
    %     my=my-imag(box(4));
    %
        %Calculating the angle of all data.
        rd=sqrt(mx.^2+my.^2);
        %the following is done to prevent divide by zero error.

        %FORMELT SKAL HER CHECKES OM NOGLE AF R = 0! FOR AT UNDGÅ DIVIDE WITH
        %ZERO.


        %Angle of point
        ang_d=acos(mx./rd);

        %Investigating the solution to chose from the angle calculation.
%         for i=1:length(ang_d)
%             if my<0
%                 ang_d(i)=-1*ang_d(i);
%             end
%             if mx<0
%                 ang_d(i)=ang_d(i)+pi;
%             end
%         end

        %Subracting the difference between the angle and the slope
        new_ang_d=ang_d-box_slope;

        new_xd=rd.*cos(new_ang_d);
        new_yd=rd.*sin(new_ang_d);

        new_data=complex(new_xd,new_yd);

%         %Plot af før efter af rotation.
%         plot(new_box)
%         hold on
%         plot(box,'r')
%         plot(new_data,'.')
%         plot(mx,my,'.r')

%-------------------------------------------------------------------------
%
%               Identification of the mean/median
%
%-------------------------------------------------------------------------
```

```
        %Test using 1/4 equidistant.

        %Width of the box.
        width=max(real(new_box))-min(real(new_box));
        hight=max(imag(new_box))-min(imag(new_box));

        part1L=0;
        part2L=0;
        part3L=0;
        part4L=0;

        part1U=0;
        part2U=0;
        part3U=0;
        part4U=0;

        count1L=1;
        count2L=1;
        count3L=1;
        count4L=1;

        count1U=1;
        count2U=1;
        count3U=1;
        count4U=1;

        %Division of the boundary-box
        %-----------------------------------------------------------------
        %             |               |               |               |
        %    1U       |       2U      |       3U      |       4U      |
        %             |               |               |               |
        %-----------------------------------------------------------------
        %             |               |               |               |
        %    1L       |       2L      |       3L      |       4L      |
        %             |               |               |               |
        %-----------------------------------------------------------------


        for i=1:length(new_data)
            if new_data(i)>min(real(new_box)) &&
new_data(i)<=min(real(new_box))+width/4;
                if imag(new_data(i))>min(imag(new_box)) &&
imag(new_data(i))<=min(imag(new_box))+hight/2;
                    part1L(count1L)=new_data(i);
                    count1L=count1L+1;
                end
                if imag(new_data(i))>min(imag(new_box))+hight/2 &&
imag(new_data(i))<=max(imag(new_box));
                    part1U(count1U)=new_data(i);
                    count1U=count1U+1;
                 end
            end
            if new_data(i)>min(real(new_box))+width/4 &&
new_data(i)<=min(real(new_box))+2*width/4;
                if imag(new_data(i))>min(imag(new_box)) &&
imag(new_data(i))<=min(imag(new_box))+hight/2;
                    part2L(count2L)=new_data(i);
                    count2L=count2L+1;
```

```
                end
                if imag(new_data(i))>min(imag(new_box))+hight/2 &&
imag(new_data(i))<=max(imag(new_box));
                    part2U(count2U)=new_data(i);
                    count2U=count2U+1;
                end
            end
            if new_data(i)>min(real(new_box))+2*width/4 &&
new_data(i)<=min(real(new_box))+3*width/4;
                if imag(new_data(i))>min(imag(new_box)) &&
imag(new_data(i))<=min(imag(new_box))+hight/2;
                    part3L(count3L)=new_data(i);
                    count3L=count3L+1;
                end
                if imag(new_data(i))>min(imag(new_box))+hight/2 &&
imag(new_data(i))<=max(imag(new_box));
                    part3U(count3U)=new_data(i);
                    count3U=count3U+1;
                end
            end
            if new_data(i)>min(real(new_box))+3*width/4 &&
new_data(i)<=min(real(new_box))+width;
                if imag(new_data(i))>min(imag(new_box)) &&
imag(new_data(i))<=min(imag(new_box))+hight/2;
                    part4L(count4L)=new_data(i);
                    count4L=count4L+1;
                end
                if imag(new_data(i))>min(imag(new_box))+hight/2 &&
imag(new_data(i))<=max(imag(new_box));
                    part4U(count4U)=new_data(i);
                    count4U=count4U+1;
                end
            end
        end



    %Checking to see if there is enough observations in the different zones
    %of the boundary box. If a zone has fewer obs than the threshold
    %min_obs requires then the zone is not updated and the previous
    %estimate is used. The funny way of counting is due to mouth being
    %defined from right to left, hvile box calculations are done from left
    %to right :s

%     %endpoints
%     if length(part1L)+length(part1U)<min_obs
%         part1L=mouth(1);
%         part1U=mouth(1);
%     end
%     if length(part4L)+length(part4U)<min_obs
%         part4L=mouth(4);
%         part4U=mouth(4);
%     end
%     Midpoints (lower)

flag1l=0;
flag1u=0;
flag2l=0;
flag2u=0;
```

```
flag3l=0;
flag3u=0;
flag4l=0;
flag4u=0;


if part1L==0
    part1L=mouth(1);
    flag1l=1;
end
if part1U==0
    part1U=mouth(1);
    flag1u=1;
end
if part2L==0
    part2L=mouth(6);
    flag2l=1;
end
if part2U==0
    part2U=mouth(2);
    flag2u=1;
end
if part3L==0
    part3L=mouth(5);
    flag3l=0;
end
if part3U==0
    part3U=mouth(3);
    flag3u=0;
end
if part4L==0
    part4L=mouth(4);
    flag4l=1;
end
if part4U==0
    part4U=mouth(4);
    flag4u=1;
end



    % % % %-----------------------------------------------------------------
    % % % %
    % % % %                                   MEAN
    % % % %
    % % % %-----------------------------------------------------------------
    % % %
    % % %     %Left and right side of mouth.
    % % %     part1=mean([part1L,part1U]);
    % % %     part4=mean([part4L,part4U]);
    % % %
    % % %     %Lower lips.
    % % %     part2L=mean(part2L);
    % % %     part3L=mean(part3L);
    % % %
    % % %     part2L=complex(real(part2L),mean([imag(part2L),imag(part3L)]));
    % % %     part3L=complex(real(part3L),mean([imag(part2L),imag(part3L)]));
    % % %
    % % %     %Upper lips.
```

```
    % % %     part2U=mean(part2U);
    % % %     part3U=mean(part3U);
    % % %
    % % %     part2U=complex(real(part2U),mean([imag(part2U),imag(part3U)]));
    % % %     part3U=complex(real(part3U),mean([imag(part2U),imag(part3U)]));
    % % %
    % % %     lipplot=[part1, part2L, part2U, part3L,part3U, part4];
    % % %
    % % %     plot(lipplot,'^')


    %-------------------------------------------------------------------------
    %
    %                               MEDIAN
    %
    %-------------------------------------------------------------------------

        %Upper lip.
        part2U=complex(median(real(part2U)),median(imag(part2U)));
        part2L=complex(median(real(part2L)),median(imag(part2L)));
        %
        part2x=median([real(part2U),real(part2L)]);

        %Lower lip
        part3U=complex(median(real(part3U)),median(imag(part3U)));
        part3L=complex(median(real(part3L)),median(imag(part3L)));
        part3x=median([real(part3U),real(part3L)]);

        %Final estimate of the lips, after averaging over x and y.
        part2LL=complex(part2x,median([imag(part2L),imag(part3L)]));
        part3LL=complex(part3x,median([imag(part2L),imag(part3L)]));

        part2UU=complex(part2x,median([imag(part2U),imag(part3U)]));
        part3UU=complex(part3x,median([imag(part2U),imag(part3U)]));

        %Left and right side of mouth.
        part1Ua=complex(median(real(part1U)),median(imag(part1U)));
        part1La=complex(median(real(part1L)),median(imag(part1L)));

part1=complex(mean([real(part1Ua),real(part1La)]),mean([imag(part1Ua),imag(part1
La)]));

        part4Ua=complex(median(real(part4U)),median(imag(part4U)));
        part4La=complex(median(real(part4L)),median(imag(part4L)));

part4=complex(mean([real(part4Ua),real(part4La)]),mean([imag(part4Ua),imag(part4
La)]));

        lipplot=[part1, part2LL, part3LL, part4, part3UU, part2UU];

    %-------------------------------------------------------------------------
    %
    %                     Rotation of the mouth to initial tilt
    %
    %-------------------------------------------------------------------------

        %finding the slope of every point.
        rm=sqrt(real(lipplot).^2+imag(lipplot).^2);
```

```
        %Angle of point
        ang_m=acos(real(lipplot)./rm);

%         %Investigating the solution to chose from the angle calculation.
%         for i=1:length(ang_m)
%             if imag(lipplot(i))<0
%                 ang_m(i)=-1*ang_m(i);
%             end
%             if real(lipplot(i))<0 && imag(lipplot(i))<0
%                 ang_m(i)=ang_m(i)+pi;
%             end
%         end

        %Subracting the difference between the angle and the slope
        new_ang_m=ang_m+box_slope;

        new_xm=rm.*cos(new_ang_m);
        new_ym=rm.*sin(new_ang_m);

        new_mouth=complex(new_xm,new_ym);
        new_mouth(7)=new_mouth(1);

        if flag1l==1
            new_mouth(1)=part1;
            new_mouth(7)=part1;
        end
        if flag1u==1
            new_mouth(1)=part1;
        end
        if flag2l==1
            new_mouth(6)=part2LL;
        end
        if flag2u==1
            new_mouth(2)=part2UU;
        end
        if flag3l==1
            new_mouth(5)=part3LL;
        end
        if flag3u==1
            new_mouth(3)=part3UU;
        end
        if flag4l==1
            new_mouth(4)=part4;
        end
        if flag4u==1
            new_mouth(4)=part4;
        end
else
    new_mouth=mouth;
end
```

13 Appendix C: jAER java Filter code

13.1 Descriptive

```
/* Descriptive.java
 *
 * Created on 14. april 2008, 14:42
 *
 * @Author: Alexander Tureczek
 *
 * This class implements descriptive statistics. takes as input a list and
 * returns a float number, containing the median of the list.
 */

package ch.unich.ini.caviar.face_track;

public class Descriptive {

    public Descriptive(){}

    //this is used to calculate the median of a list.
    public float Median(float[] list)
    {
        float median = list.length;
        short index=0;


        //if the list is even length
        if (median%2==0)
        {
            median = (float) list[(list.length/2)-1];
            median = (float) list[(list.length/2)]+median;
            median = median/2;
        }
        //if the list is odd length
        else
        {
            index = (short) ((list.length + 1) / 2 - 1);
            median = (short) list[index];
        }

        return (float) median;
    }
}
```

13.2 FaceTrack

```
/* FaceTrack.java
 *
 * Created on 14. april 2008, 18.03
 *
 * @Author: Alexander Tureczek
 *
 * This class is controlling the flow of the filter, and initializing all
 * variables and setting the GUI values. Also used to collect the data from
the
 * silicon retina.
 *
 */


package ch.unich.ini.caviar.face_track;

import ch.unizh.ini.caviar.chip.*;
import ch.unizh.ini.caviar.chip.AEChip;
import ch.unizh.ini.caviar.event.*;
import ch.unizh.ini.caviar.event.EventPacket;
import ch.unizh.ini.caviar.eventprocessing.EventFilter2D;
import java.util.*;
import java.lang.Math.*;
import ch.unizh.ini.caviar.eventprocessing.*;
import ch.unizh.ini.caviar.eventprocessing.filter.SubSampler;
import ch.unizh.ini.caviar.graphics.FrameAnnotater;
import ch.unizh.ini.caviar.util.VectorHistogram;
import java.awt.Graphics2D;
import java.awt.geom.Point2D;
import javax.media.opengl.GL;
import javax.media.opengl.GLAutoDrawable;


public class FaceTrack extends EventFilter2D implements FrameAnnotater,
Observer {

    //final int NEVENTS = 1000;
    protected int NEVENTS = getPrefs().getInt("NEVENTS", 1000);
    {setPropertyTooltip("NEVENTS","Number of events to buffer before
updating the model");}

    protected int LEDBOX = getPrefs().getInt("LEDBOX",5);
    {setPropertyTooltip("LEDBOX","Sets the size of the allowed movement of
LED");}

    //Collecting data in two vectors x and y.
    int[] x_data = new int[NEVENTS];
    int[] y_data = new int[NEVENTS];
    float[] LED = new float[5];
    float[] LEDold = new float[5];
    //float[][] hori_meanS = new float[8][2];
    float f_dist;
    float[][] new_hori = new float[20][2];
    float[][] pre_mouth = new float[7][2];
    float[][] mouth_boundary = new float[5][2];
```

```
52      int face_end = 7; //Face has 8 landmarks but JAVA counts from 0.
53
54      //Constructor.
55      public FaceTrack(AEChip chip) {
56          super(chip);
57          initFilter();
58          resetFilter();
59      }
60
61      @Override
62      public Object getFilterState() {
63          return null;
64      }
65
66      //Resetting the filter, is only done when the filter is initialized.
67      @Override
68      public void resetFilter() {
69          //Defining variables used by the filter
70
71          //initializing the face model.
72          HorizontalFaceModel model = new HorizontalFaceModel();
73          float[][] face =(float[][]) model.InitialFace();
74
75          //calculating the size of the face model, used for scaling to the
LED.
76          HorizontalFaceModel dist = new HorizontalFaceModel();
77          f_dist =(float) dist.FACEdist(face);
78
79          //initial guess of the LED, calculated in matlab.
80          int[] LED1 = {35, 80};
81          int[] LED2 = {105, 80};
82
83          //Calculating the distance between the LED
84          int x = (int) ((LED2[0] - LED1[0]) * (LED2[0] - LED1[0]));
85          int y = (int) ((LED2[1] - LED1[1]) * (LED2[1] - LED1[1]));
86          //calculation of distance using standard foormula.
87          float LEDdist = (float) Math.sqrt(x+ y);
88
89          LEDold[0] = LED1[0];
90          LEDold[1] = LED1[1];
91          LEDold[2] = LED2[0];
92          LEDold[3] = LED2[1];
93          LEDold[4] = (float) LEDdist;
94
95          //Defining the boundary of the mouth. the boundary points are
surrounding
96          //the mouth, that is the 7 last observations in hori_meanS.java
97          float minx = face[8][0];
98          float miny = face[8][1];
99          float maxx = face[8][0];
100         float maxy = face[8][1];
101
102         for (int i = 8; i < face.length; i++) {
103             //finding minimum for the bounadary box around the emouth.
104             if (face[i][0] < minx) {
105                 minx = face[i][0];
106             }
107             if (face[i][1] < miny) {
108                 miny = face[i][1];
```

```
109             }
110             //finding maximum for the bounadary box around the emouth.
111             if (face[i][0] > maxx) {
112                 maxx = face[i][0];
113             }
114             if (face[i][1] > maxy) {
115                 maxy = face[i][1];
116             }
117         }
118 
119 // ---------------------------------------------------------------------
120 //
121 //      Check disse udregninger igen, jeg kan ikke gennemskue dem
122 //
123 // ---------------------------------------------------------------------
124         float x_add = (float) (minx - 0.12);
125         float y_add = (float) (miny + 0.42);
126 
127 
128         //X-coordinates of the boundary box
129         mouth_boundary[0][0] = minx - x_add;
130         mouth_boundary[1][0] = maxx + x_add;
131         mouth_boundary[2][0] = maxx + x_add;
132         mouth_boundary[3][0] = minx - x_add;
133         mouth_boundary[4][0] = minx - x_add;
134 
135         //Y-coordinates of the boundary box
136         mouth_boundary[0][1] = miny - y_add;
137         mouth_boundary[1][1] = miny - y_add;
138         mouth_boundary[2][1] = maxy + y_add;
139         mouth_boundary[3][1] = maxy + y_add;
140         mouth_boundary[4][1] = miny - y_add;
141 
142 //---------------------------------------------------------------------
143 //
144 //          Creating initial mouth and face
145 //
146 //---------------------------------------------------------------------
147 
148         //Creating combined face, mouth and boundary vector
149         for (int i = (int) 0; i < face.length + mouth_boundary.length; i++)
{
150             if (i < face.length)
151             {
152                 new_hori[i][0] = face[i][0];
153                 new_hori[i][1] = face[i][1];
154             }
155             else
156             {
157                 new_hori[i][0] = mouth_boundary[i - face.length][0];
158                 new_hori[i][1] = mouth_boundary[i - face.length][1];
159             }
160             if (i > face_end && i < face.length)
161             {
162                 pre_mouth[i - face_end - 1][0] = face[i][0];
163                 pre_mouth[i - face_end - 1][1] = face[i][1];
164             }
165         }
166 
```

```
167         //Setting the end point equal starting point to make sure the mouth
is
168         //closed
169         pre_mouth[6][0] = pre_mouth[0][0];
170         pre_mouth[6][1] = pre_mouth[0][1];
171
172         for (int index=0; index<new_hori.length; index++)
173         {
174             new_hori[index][0]=new_hori[index][0]+LEDold[0];
175             new_hori[index][1]=new_hori[index][1]+LEDold[1];
176         }
177
178     }
179
180     //@Override
181     public void initFilter() {
182         resetFilter();
183     }
184
185     //Parameter setting in the GUI, Setting NEVENTS and LEDBOX
186     public int getNEVENTS() {
187         return this.NEVENTS;
188     }
189
190     //Setting the NEVENTS parameters in the parameter box in the GUI.
191     public void setNEVENTS(final int NEVENTS) {
192         getPrefs().putInt("FaceTracker.NEVENTS",NEVENTS);
193         support.firePropertyChange("NEVENTS",this.NEVENTS,NEVENTS);
194         this.NEVENTS = NEVENTS;
195     }
196
197     //Getting the LEDBOX parameters in the parameter box in the GUI.
198     public int getLEDBOX() {
199         return this.LEDBOX;
200     }
201
202     //Setting the LEDBOX parameters in the parameter box in the GUI.
203     public void setLEDBOX(final int LEDBOX) {
204         getPrefs().putInt("FaceTracker.LEDBOX",LEDBOX);
205         support.firePropertyChange("LEDBOX",this.LEDBOX,LEDBOX);
206         this.LEDBOX = LEDBOX;
207     }
208
209     public void update(Observable o, Object arg) {
210         initFilter();
211     }
212     int numEventsCollected = 0;
213
214     //Method for collecting data from the retina.
215     //@Override
216     public EventPacket<?> filterPacket(EventPacket<?> in) {
217
218         if (!filterEnabled) {
219             return in;
220         } //Check to avoid always running filter.
221         if (enclosedFilter != null) {
222             in = enclosedFilter.filterPacket(in);
223         }
224
```

```
225         for (Object e : in) {
226             BasicEvent i = (BasicEvent) e;
227             x_data[numEventsCollected] = (int)i.x;
228             y_data[numEventsCollected] = (int)i.y;
229             numEventsCollected++;
230             if (numEventsCollected == NEVENTS) {
231                 processEvents();
232                 numEventsCollected = 0;
233             }
234         }
235 
236         return in;
237     }
238 
239     private void processEvents() {
240 
241         //Calculating the updated estimates of the LED positions, and the
distance between the LED.
242         //initializing the followLED class, used for updating the LED
position
243         //shape and size of the face.
244         FollowLED shapeUpdate = new FollowLED();
245         HorizontalFaceModel fdist = new HorizontalFaceModel();
246         Mund mouth = new Mund();
247 
248         //LED is an array of 5 elements with information about the Position
of the LED and distance between them.
249         LED = shapeUpdate.movLED(LEDold, x_data, y_data, LEDBOX);
250 
251         //Updating the shape of the model.
252         new_hori = shapeUpdate.rotator(LED, f_dist, new_hori, LEDold);
253 
254         //Updating the FACEdist.
255         f_dist = fdist.FACEdist(new_hori);
256 
257         //Updating the mouth
258         //pre_mouth = mouth.rotateMouth(x_data, y_data, new_hori, pre_mouth,
mouth_boundary);
259 
260         LEDold=LED;
261     }
262 
263     //Plotting of the vectors, new_hori(0:7), pre_mouth and LED.
264     public void annotate(Graphics2D g) {}
265 
266     //Method for annotating arrays in the GUI, using openGl
267     synchronized public void annotate(GLAutoDrawable drawable) {
268         final float LINE_WIDTH=6f; // in pixels
269         if(!isFilterEnabled()) return;
270         GL gl=drawable.getGL(); // when we get this we are already set up
with scale 1=1 pixel, at LL corner
271         if(gl==null){
272             log.warning("null GL in Face_Track.annotate");
273             return;
274         }
275 
276         float[] rgb=new float[4];
277         rgb[0]=1;
278         gl.glPushMatrix();
```

```
279
280         gl.glColor3fv(rgb,0);
281         gl.glLineWidth(LINE_WIDTH);
282         gl.glPointSize(LINE_WIDTH);
283         gl.glBegin(GL.GL_POINTS);
284         for (int index=0; index<=new_hori.length-1; index++)
285         {
286             //gl.glVertex2f(LED[index][1],LED[index][0]);
287             gl.glVertex2f(new_hori[index][0],new_hori[index][1]);
288
289                 if (index>face_end && index<(new_hori.length-5))
290                 {
291                     gl.glVertex2f(pre_mouth[index-8][0],pre_mouth[index-
8][1]);
292                 }
293         }
294         gl.glEnd();
295         gl.glPopMatrix();
296
297     }
298
299     //Empty method for plotting, needs to implemented but not used.
300     public void annotate(float[][][] frame) {}
301
302     //Used for standardizing arrays removing the LED1 influence.
303     public void standard(float[][]new_hori)
304         {
305             for (int index=0; index<new_hori.length; index++)
306             {
307                 new_hori[index][0]=new_hori[index][0]-LED[0];
308                 new_hori[index][1]=new_hori[index][1]-LED[1];
309             }
310
311     }
312 }
```

13.3 HorizontalFaceModel

```
/* HorizontalFaceModel.java
 *
 * Created on 20. april 2008, 16:14
 *
 * Author: Alexander Tureczek
 *
/* This class is governing the model used in the facetracker. The face has been
 meancentered and rotated to up-right position. It has been converted from complex
 notation in MatLab xy notation in JAVA. The rotation and alligning methods has
 not been implemented in JAVA and thus a new model has to be made in MatLab with
 the functions meancent and rotation. The size of the model is recommended to be
 of an even number of landmarks.
 */

package ch.unich.ini.caviar.face_track;
import java.lang.Math;


public class HorizontalFaceModel {

    public HorizontalFaceModel(){}

    //This contains the coordinates of the initial model, rotated but NOT scaled.
        public float[][] InitialFace()
        {

            float[][] meanshape = new float[15][2];
            //This is the x-coordinates of the model
            meanshape[0][0]=0f;
            meanshape[1][0]=0.0208f;
            meanshape[2][0]=0.0581f;
            meanshape[3][0]=0.2440f;
            meanshape[4][0]=0.4264f;
            meanshape[5][0]=0.6118f;
            meanshape[6][0]=0.6474f;
            meanshape[7][0]=0.6605f;

            //Landmarks for the mouth, x-coordinates
            meanshape[8][0]=0.1823f;
            meanshape[9][0]=0.2728f;
            meanshape[10][0]=0.3530f;
            meanshape[11][0]=0.4551f;
            meanshape[12][0]=0.3694f;
            meanshape[13][0]=0.2615f;
            meanshape[14][0]=0.1823f;

            //This is the y-coordinates of the model
            meanshape[0][1]=-0f;
            meanshape[1][1]=-0.2156f;
```

```
50              meanshape[2][1]=-0.3668f;
51              meanshape[3][1]=-0.5600f;
52              meanshape[4][1]=-0.5561f;
53              meanshape[5][1]=-0.3774f;
54              meanshape[6][1]=-0.2131f;
55              meanshape[7][1]=-0f;
56              //Landmarks for the mouth, y-coordinates
57              meanshape[8][1] =-0.3183f;
58              meanshape[9][1] =-0.3044f;
59              meanshape[10][1]=-0.3003f;
60              meanshape[11][1]=-0.3176f;
61              meanshape[12][1]=-0.3843f;
62              meanshape[13][1]=-0.3843f;
63              meanshape[14][1]=-0.3183f;
64
65              return (float[][]) meanshape;
66      }
67
68
69
70      //This method calculates the distance between the (1,1) and (8,1), this
is
71      //used to calculate the scaling of the model to the LED.
72      public float FACEdist(float meanshape[][])
73      {
74          float x,y, dist;
75
76          x=(float) (meanshape[7][0]-meanshape[0][0])*(meanshape[7][0]-
meanshape[0][0]);
77          y=(float) (meanshape[7][1]-meanshape[0][1])*(meanshape[7][1]-
meanshape[0][1]);
78
79          //calculation of distance using standard foormula.
80          dist=(float) Math.sqrt(x+y);
81
82          return dist;
83      }
84
85  }
```

13.4 Mund.java

```
/*  Mund.java
 *
 * Created on 13. juni 2008, 14:15
 *
 * @Author: Alexander Tureczek
 *
 * Governs the mouth and the deformation of it. It takes as input the data
 * vector and the model. Though rotation and scaling the model is deformed
to
 * fit the data.
 */

package ch.unich.ini.caviar.face_track;

import java.lang.Math.*;
import java.util.ArrayList;

public class Mund
{

    //empty constructor
    public Mund(){}

    public float[][] rotateMouth(int[] x, int[] y, float[][] hori_meanS,
float[][] pre_mouth, float[][] mouth_boundary)
    {
        //identifying the boundary box.
        int count=0;
        float[][] box = new float[5][2];

        for (int index=15; index<hori_meanS.length; index++)
        {
            box[count][0]=hori_meanS[index][0];
            box[count][1]=hori_meanS[index][1];
            count++;
        }
        //calculating the tilt of the mouth_boundary box.

        float box_slope = (box[2][1]-box[3][1])/(box[2][0]-box[3][0]);

        float[] r = new float[5];
        float[] ang = new float[5];
        float[] new_ang = new float[5];
        float[][] new_box = new float[5][2];

        for (int index=0; index<box.length; index++)
        {
            //calculating the argument of every point. using normal a^2+b^2
= c^2

            r[index] = (float)
Math.sqrt(Math.pow(box[index][0],2)+Math.pow(box[index][1],2));

            //Calculating the modulo of the boundary.

```

```
52              ang[index] = (float) Math.acos(box[index][0]/r[index]);
53
54              //Subracting the difference between the angle and the slope
55
56              new_ang[index] = ang[index]-box_slope;
57
58              //calculating the final horisontal box.
59
60              new_box[index][0] = (float) (r[index]*Math.cos(new_ang[index]));
61              new_box[index][1] = (float) (r[index]*Math.sin(new_ang[index]));
62          }
63
64  //  ---------------------------------------------------------------------
65  //
66  //              Finding relevant data and doing Rotation.
67  //
68  //  ---------------------------------------------------------------------
69
70          //Finding the relevant data.  -> data within the mouth_boundary
71
72          float[] mx = new float[x.length];
73          float[] my = new float[x.length];
74          count=0;
75
76          //finding the smallest and largest x and y, to find a box that
surrounds the mouth.
77          float minx = box[0][0];
78          float miny = box[0][1];
79          float maxx = box[0][0];
80          float maxy = box[0][1];
81
82          for (int index=0; index < box.length; index++) {
83              //finding minimum for the bounadary box around the emouth.
84              if (box[index][0] < minx)
85              {
86                  minx = box[index][0];
87              }
88              if (box[index][1] < miny)
89              {
90                  miny = box[index][1];
91              }
92              //finding maximum for the bounadary box around the emouth.
93              if (box[index][0] > maxx)
94              {
95                  maxx = box[index][0];
96              }
97              if (box[index][1] > maxy)
98              {
99                  maxy = box[index][1];
100             }
101         }
102         //finding all relevant data, e.g. data inside the boundary box.
103         count=0;
104         for (int index=0; index<x.length; index++)
105         {
106             if ((x[index]>=minx) && (x[index]<=maxx) && (y[index]>=miny) &&
(y[index]<=maxy))
107             {
108                     mx[count]=x[index];
```

```
109                     my[count]=y[index];
110                     count++;
111             }
112         }
113 
114         //if relevant data is found, we analyse this.
115         if (count>0)
116         {
117             //calculating the angle for all data. Used to rotate
118 
119             //Modulo for data
120             float[] rd = new float[count];
121             // Argument for data
122             float[] ang_d = new float[count];
123             float[][] new_data = new float[count][2];
124             float[] new_ang_d = new float[count];
125 
126 
127             for (int index=0; index<count; index++)
128             {
129 
rd[index]=(float)(Math.sqrt(Math.pow(mx[index],2)+Math.pow(my[index],2)));
130 
131                 ang_d[index]=(float)(Math.acos(mx[index]/rd[index]));
132 
133                 //Here is not included check for solution due to float
solution.
134 
135                 //Subtracting the difference in angles
136                 new_ang_d[index] = ang_d[index]-box_slope;
137 
138                 //calculating the final horisontal data.
139                     new_data[index][0] =
(float)(rd[index]*Math.cos(new_ang_d[index]));
140                     new_data[index][1] =
(float)(rd[index]*Math.sin(new_ang_d[index]));
141             }
142 
143 
144 
145 
146 //------------------------------------------------------------------------
147 //
148 //              Identification of # obs in each cell.
149 //
150 //------------------------------------------------------------------------
151 
152         //Test using 1/4 equidistant.
153 
154         //Width of the box.
155         float width=new_box[1][0]-new_box[0][0];
156         float height=new_box[3][1]-new_box[0][1];
157 
158         int count1L=0;
159         int count2L=0;
160         int count3L=0;
161         int count4L=0;
162 
163         int count1U=0;
```

```
164         int count2U=0;
165         int count3U=0;
166         int count4U=0;
167
168         //This is initiating each cell in the boundary (c.f report for cell
division)
169
170         //this for-loop is used to determine the number of observations in
each cell.
171         //this is used in the next for-loop to define the vector for each
cell.
172          for (int index=0; index<new_data.length; index++)
173         {
174             //Left cell calculation. First finding the x-data and then
finding the y-data.
175             if (new_data[index][0]>new_box[0][0] &&
new_data[index][0]<=new_box[0][0]+width/4)
176             {
177                 if (new_data[index][1]>new_box[0][1] &&
new_data[index][1]<=new_box[0][1]+height/2)
178                 {
179                     count1L++;
180                 }
181                 if (new_data[index][1]>new_box[0][1]+height/2 &&
new_data[index][1]<=new_box[3][1])
182                 {
183                     count1U++;
184                 }
185             }
186             //Cell 2 from the left.
187              if (new_data[index][0]>new_box[0][0]+width/4 &&
new_data[index][0]<=new_box[0][0]+2*width/4)
188             {
189                 if (new_data[index][1]>new_box[0][1] &&
new_data[index][1]<=new_box[0][1]+height/2)
190                 {
191                     count2L++;
192                 }
193                 if (new_data[index][1]>new_box[0][1]+height/2 &&
new_data[index][1]<=new_box[3][1])
194                 {
195                     count2U++;
196                 }
197             }
198             //third cell from the left.
199             if (new_data[index][0]>new_box[0][0]+2*width/4 &&
new_data[index][0]<=new_box[0][0]+3*width/4)
200             {
201                 if (new_data[index][1]>new_box[0][1] &&
new_data[index][1]<=new_box[0][1]+height/2)
202                 {
203                     count3L++;
204                 }
205                 if (new_data[index][1]>new_box[0][1]+height/2 &&
new_data[index][1]<=new_box[3][1])
206                 {
207                     count3U++;
208                 }
209             }
```

```
210             //Right cell
211             if (new_data[index][0]>new_box[0][0]+3*width/4 &&
new_data[index][0]<=new_box[0][0]+width)
212             {
213                 if (new_data[index][1]>new_box[0][1] &&
new_data[index][1]<=new_box[0][1]+height/2)
214                 {
215                     count4L++;
216                 }
217                 if (new_data[index][1]>new_box[0][1]+height/2 &&
new_data[index][1]<=new_box[3][1])
218                 {
219                     count4U++;
220                 }
221             }
222         }
223
224         //knowing the number of observations in each cell, we define the
arrays for each cell.
225         float[][] part1L = new float[count1L][2];
226         float[][] part1U = new float[count1U][2];
227         float[][] part2L = new float[count2L][2];
228         float[][] part2U = new float[count2U][2];
229         float[][] part3L = new float[count3L][2];
230         float[][] part3U = new float[count3U][2];
231         float[][] part4L = new float[count4L][2];
232         float[][] part4U = new float[count4U][2];
233
234         //Resetting the counting.
235         count1L=0;
236         count2L=0;
237         count3L=0;
238         count4L=0;
239
240         count1U=0;
241         count2U=0;
242         count3U=0;
243         count4U=0;
244
245
246         for (int index=0; index<new_data.length; index++)
247         {
248             //Left cell calculation. First finding the x-data and then
finding the y-data.
249             if (new_data[index][0]>new_box[0][0] &&
new_data[index][0]<=new_box[0][0]+width/4)
250             {
251                 if (new_data[index][1]>new_box[0][1] &&
new_data[index][1]<=new_box[0][1]+height/2)
252                 {
253                     part1L[count1L][0] = new_data[index][0];
254                     part1L[count1L][1] = new_data[index][1];
255                     count1L++;
256                 }
257                 if (new_data[index][1]>new_box[0][1]+height/2 &&
new_data[index][1]<=new_box[3][1])
258                 {
259                     part1U[count1U][0] = new_data[index][0];
260                     part1U[count1U][1] = new_data[index][1];
```

```
261                     count1U++;
262                 }
263             }
264             //Cell 2 from the left.
265              if (new_data[index][0]>new_box[0][0]+width/4 &&
new_data[index][0]<=new_box[0][0]+2*width/4)
266             {
267                 if (new_data[index][1]>new_box[0][1] &&
new_data[index][1]<=new_box[0][1]+height/2)
268                 {
269                     part2L[count2L][0] = new_data[index][0];
270                     part2L[count2L][1] = new_data[index][1];
271                     count2L++;
272                 }
273                 if (new_data[index][1]>new_box[0][1]+height/2 &&
new_data[index][1]<=new_box[3][1])
274                 {
275                     part2U[count2U][0] = new_data[index][0];
276                     part2U[count2U][1] = new_data[index][1];
277                     count2U++;
278                 }
279             }
280             //third cell from the left.
281             if (new_data[index][0]>new_box[0][0]+2*width/4 &&
new_data[index][0]<=new_box[0][0]+3*width/4)
282             {
283                 if (new_data[index][1]>new_box[0][1] &&
new_data[index][1]<=new_box[0][1]+height/2)
284                 {
285                     part3L[count3L][0] = new_data[index][0];
286                     part3L[count3L][1] = new_data[index][1];
287                     count3L++;
288                 }
289                 if (new_data[index][1]>new_box[0][1]+height/2 &&
new_data[index][1]<=new_box[3][1])
290                 {
291                     part3U[count3U][0] = new_data[index][0];
292                     part3U[count3U][1] = new_data[index][1];
293                     count3U++;
294                 }
295             }
296             //Right cell
297             if (new_data[index][0]>new_box[0][0]+3*width/4 &&
new_data[index][0]<=new_box[0][0]+width)
298             {
299                 if (new_data[index][1]>new_box[0][1] &&
new_data[index][1]<=new_box[0][1]+height/2)
300                 {
301                     part4L[count4L][0] = new_data[index][0];
302                     part4L[count4L][1] = new_data[index][1];
303                     count4L++;
304                 }
305                 if (new_data[index][1]>new_box[0][1]+height/2 &&
new_data[index][1]<=new_box[3][1])
306                 {
307                     part4U[count4U][0] = new_data[index][0];
308                     part4U[count4U][1] = new_data[index][1];
309                     count4U++;
310                 }
```

```
311             }
312         }
313 
314 //  ------------------------------------------------------------------------
315 //
316 //                  Calculating the mean in each cell
317 //
318 //  ------------------------------------------------------------------------
319 
320         //Initializing the sorting and median methods.
321         Sorting heap = new Sorting();
322         Descriptive median = new Descriptive();
323 
324         //Calculating median x and y in cell 2L and 2U
325         part2L =(float[][]) heap.medianHeap(part2L);
326         part2U =(float[][]) heap.medianHeap(part2U);
327 
328         //putting the median x value from cell 2L and 2U in part2x
329         float part2x = (part2L[0][0]+part2U[0][0])/2;
330 
331         //Calculating median x and y in cell 3L and 3U
332         part3L = (float[][]) heap.medianHeap(part3L);
333         part3U = (float[][]) heap.medianHeap(part3U);
334 
335         //putting the median x value from cell 3L and 3U in part3x
336         float part3x = (float)(part3L[0][0]+part3U[0][0])/2;
337 
338 
339         //Averaging over 2L and 3L
340         float part23L = (float)(part2L[0][1]+part3L[0][1])/2;
341         float part23U = (float)(part2U[0][1]+part3U[0][1])/2;
342 
343         //final estimate of 2L and 3L
344         part2L[0][0] = part2x;
345         part2L[0][1] = part23L;
346 
347         part3L[0][0] = part3x;
348         part3L[0][1] = part23L;
349 
350         //final estimate of 2U and 3U
351         part2U[0][0] = part2x;
352         part2U[0][1] = part23U;
353 
354         part3L[0][0] = part3x;
355         part3L[0][1] = part23U;
356 
357 
358         //Estimating 1L and 1U
359         part1L = heap.medianHeap(part1L);
360         part1U = heap.medianHeap(part1U);
361 
362         float part1x = (float)(part1L[0][0]+part1U[0][0])/2;
363         float part1y = (float)(part1L[0][1]+part1U[0][1])/2;
364 
365         part1L[0][0]=part1x;
366         part1U[0][0]=part1x;
367 
368         part1L[0][1]=part1y;
369         part1U[0][1]=part1y;
```

```
370
371
372         //Estimating 4L and 4U
373         part4L = (float[][]) (heap.medianHeap(part4L));
374         part4U = (float[][]) (heap.medianHeap(part4U));
375
376         float part4x = (float)(part4L[0][0]+part4U[0][0])/2;
377         float part4y = (float)(part4L[0][1]+part4U[0][1])/2;
378
379         part4L[0][0]=part4x;
380         part4U[0][0]=part4x;
381
382         part4L[0][1]=part4y;
383         part4U[0][1]=part4y;
384
385         // collecting all estimates in 1 array.
386         float[][] new_mouth = new float[7][2];
387         new_mouth[0][0] = part1x;
388         new_mouth[1][0] = part2x;
389         new_mouth[2][0] = part3x;
390         new_mouth[3][0] = part4x;
391         new_mouth[4][0] = part3x;
392         new_mouth[5][0] = part2x;
393         new_mouth[6][0] = part1x;
394
395         new_mouth[0][1] = part1y;
396         new_mouth[1][1] = part23L;
397         new_mouth[2][1] = part23L;
398         new_mouth[3][1] = part4y;
399         new_mouth[4][1] = part23U;
400         new_mouth[5][1] = part23U;
401         new_mouth[6][1] = part1y;
402
403
404 // --------------------------------------------------------------------
405 //
406 //              Rotating the estimate bock to original tilt
407 //
408 // --------------------------------------------------------------------
409
410         //finding the slope of every point.
411         //calculating the angle for all data. Used to rotate
412
413         //Modulo for data
414         float[] rm = new float[new_mouth.length];
415         // Argument for data
416         float[] new_ang_m = new float[new_mouth.length];
417
418         //float[][] pre_mouth = new float[new_mouth.length][2];
419         float[] new_m = new float[new_mouth.length];
420         float[] ang_m = new float[new_mouth.length];
421
422
423         for (int index=0; index<new_mouth.length; index++)
424         {
425
rm[index]=(float)(Math.sqrt(Math.pow(new_mouth[index][0],2)+Math.pow(new_mouth[i
ndex][1],2)));
426
```

```
427             ang_m[index]=(float)(Math.acos(new_mouth[index][0]/rm[index]));
428 
429             //Subtracting the difference in angles
430             new_ang_m[index] =(float) ang_m[index]+box_slope;
431 
432             //calculating the final horisontal data.
433             new_mouth[index][0] =
(float)(rm[index]*Math.cos(new_ang_m[index]));
434             new_mouth[index][1] =
(float)(rm[index]*Math.sin(new_ang_m[index]));
435         }
436 
437         if (count1L==0 || count1U==0)
438         {
439             new_mouth[0][0] = pre_mouth[0][0];
440             new_mouth[0][1] = pre_mouth[0][1];
441 
442             new_mouth[6][0] = pre_mouth[0][0];
443             new_mouth[6][1] = pre_mouth[0][1];
444         }
445 
446         if (count2L==0)
447         {
448             new_mouth[5][0] = pre_mouth[5][0];
449             new_mouth[5][1] = pre_mouth[5][1];
450         }
451 
452         if (count3L==0)
453         {
454             new_mouth[4][0] = pre_mouth[4][0];
455             new_mouth[4][1] = pre_mouth[4][1];
456         }
457 
458         if (count4L==0 || count4U==0)
459         {
460             new_mouth[3][0] = pre_mouth[3][0];
461             new_mouth[3][1] = pre_mouth[3][1];
462         }
463 
464         if (count3U==0)
465         {
466             new_mouth[2][0] = pre_mouth[2][0];
467             new_mouth[2][1] = pre_mouth[2][1];
468         }
469 
470         if (count2U==0)
471         {
472             new_mouth[1][0] = pre_mouth[1][0];
473             new_mouth[1][1] = pre_mouth[1][1];
474         }
475 
476         return new_mouth;
477         }
478 
479     return pre_mouth;
480     }
481 }
```

13.5 Sorting

```
  1 /* Sorting.java
  2  *
  3  * Created on 14. april 2008, 12:46
  4  *
  5  * Author Alexander Tureczek
  6  *
  7  *This file implements the selection sort algorithm. The selectionSort
method is
  8  *copied from Lewis/Loftus "Java, software solutions". the algorithm is
sorting
  9  *an array of numbers in O(n^2) time.
 10  *
 11  *Also implemented is the Heap Sort algorithm that sorts any vector in
n*lg(n)
 12  *time.
 13  *
 14  */
 15
 16 package ch.unich.ini.caviar.face_track;
 17 import java.lang.Math;
 18
 19
 20 public class Sorting
 21 {
 22     public Sorting(){}
 23
 24     //Contains an implementation of the selection sort algorithm.
 25     public float[] selectionSort(float[] numbers)
 26     {
 27         float temp;
 28         short min;
 29
 30         //running through the elements and moving the smaller to the left.
 31         for (short index=0; index < numbers.length-1; index++)
 32         {
 33             min = index;
 34             for (short scan =(short) (index + 1); scan < numbers.length;
scan++)
 35             {
 36                 if (numbers[scan]<numbers[min])
 37                     min = scan;
 38
 39                 //Swaping values.
 40                 temp=(float) numbers[min];
 41                 numbers[min]=(float) numbers[index];
 42                 numbers[index]=(float) temp;
 43             }
 44         }
 45     return numbers;
 46     }
 47
 48
 49     //prepares a nx2 array for heap sort row vise.
 50     public float[][] medianHeap(float[][] list)
 51     {
```

```
52          float[] list_a = new float[list.length];
53          float[] list_b = new float[list.length];
54
55          //initializing the median calculation.
56          Descriptive median = new Descriptive();
57          System.out.println(list.length);
58
59          //If the list is empty then the list is returned with zero elements.
60          if (list.length==0)
61          {
62              float[][] median_list = new float[list.length+1][2];
63              median_list[0][0] = 0;
64              median_list[0][1] = 0;
65              return (float[][]) median_list;
66          }
67          //else the list is sorted and the median is calculated through a
call to
68          //Descriptive class.
69          else
70          {
71              float[][] median_list = new float[list.length][2];
72
73              for (short index=0; index<list.length; index++)
74              {
75                  list_a[index] =(float) list[index][0];
76                  list_b[index] =(float) list[index][1];
77              }
78              list_a = (float[]) heapSort(list_a);
79              float med_x =(float) median.Median(list_a);
80
81              list_b = heapSort(list_b);
82              float med_y = median.Median(list_b);
83
84              median_list[0][0] = med_x;
85              median_list[0][1] = med_y;
86
87          return (float[][]) median_list;
88          }
89      }
90
91      // Contains an implementation of the heapsort algorithm.T.H. Cormen
et.al,
92      //"Introduction to algorithms"
93      public float[] heapSort(float[] numbers)
94      {
95          float[] A=numbers;
96          short heap_size=(short) A.length;
97          float temp;
98          A=build_heap(A);
99
100         for (short index=(short)(A.length-1); index>=1; index--)
101         {
102             temp=(float) A[0];
103             A[0]=A[index];
104             A[index]=(float) temp;
105
106             heap_size=(short) (heap_size - 1);
107             heapify(A,(short)0,heap_size);
108         }
```

```
109         return A;
110     }
111 
112     //Building the heap that is being sorted with heapify.
113     private float[] build_heap(float[] A)
114     {
115         short heap_size=(short)A.length;
116         for (short index = (short) Math.floor(A.length/2); index>=0; index--
)
117         {
118             A=heapify(A,index,heap_size);
119         }
120 
121         return A;
122     }
123 
124     //This is an implementation of the Max-heapify p.130 T.H. Cormen et.al,
125     //"Introduction to algorithms"
126     private float[] heapify(float[] A, short i, short heap_size)
127     {
128         //creating children
129         //Left, corrected for 0 indices
130         short l=(short) (2 * i + 1);
131         //Right, corrected for 0 indices
132         short r=(short) (2 * i + 2);
133 
134         short largest=0;
135 
136         //int heap_size = A.length;
137         if (l<=heap_size-1 && A[l]>A[i])
138         {
139             largest=l;
140         }
141         else
142             largest=i;
143 
144         if (r<=heap_size-1 && A[r]>A[largest])
145         {
146             largest=r;
147         }
148         if (largest!=i)
149         {
150             float temp=(float) A[i];
151             A[i]=(float) A[largest];
152             A[largest]=(float) temp;
153             heapify(A,largest,heap_size);
154         }
155         return A;
156     }
157 }
```