Facial Feature Detection Using the Hough Transform

Victor Zhou

St. Mark's School of Texas
Dallas, Texas

Research done at Total Wire Corp. of Plano Texas during summer 2012.

Special thanks to Mr. Gary Zhou and Mr. Douglas Belli.

Table of Contents:

1. Introduction

One of the oldest yet still very intriguing problems in Computer Science is the question of how to detect human faces in images. Most well-known face-detection algorithms today, including the famous Viola-Jones method, target coarse facial features and are not naturally suitable for fine facial feature detection. My research aims to develop a way to detect these finer features. Such automatic detectors would have many useful applications. For instance, a camera that can track the fine movements of eyeballs, eyelids, eyebrows, or other facial features could potentially help a machine understand human emotions. In some cases, such a machine could also serve as a useful aid for people with various disabilities. In this paper, the specific problem I choose to investigate is the real-time detection of eyes on a human face.

The Hough Transform offers a method to begin attacking this problem. The Hough Transform, first developed by Paul Hough but later extended by Richard Duda and Peter Hart, is an excellent feature extraction technique. The idea behind it is as follows: when searching an image for a certain shape, each pixel in an image can "vote" for a set of potential locations of the aforementioned shape. For example, in Figure 1, if the algorithm is searching for circles of a certain size, pixel P would vote for circles whose centers lie on the dashed ring. For instance, the circle that is centered at point Q would get a vote from pixel P. If there actually is a circle in the image centered at Q, then every point on that circle would vote for center Q, and in the end center Q would have a lot of votes, implying a high likelihood of a circle centered on it. Similarly, a point that is not actually the center of a circle would only receive votes from some random pixels.

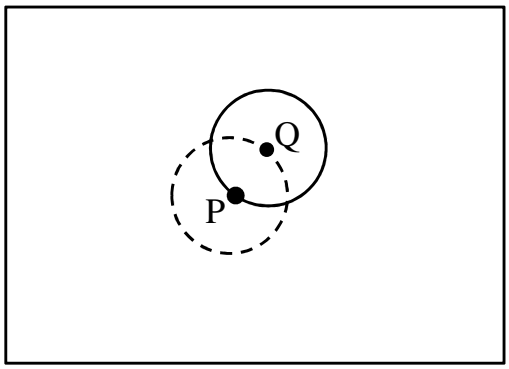

Figure 1. The Hough Transformation concept

My approach to the problem of face and feature detection is relatively unorthodox because, as mentioned above, most face detectors today use coarse, block features combined with a learning algorithm to "teach" the computer how to classify faces. For example, the Viola-Jones face detector first uses the integral image to evaluate 4 different rectangular features types. Then, using a variant of a learning algorithm named AdaBoost, it selects the best features, trains classifiers to use them, and finally combines all these classifiers into a cascade architecture. I conjectured that addressing this problem

from the different angle of Hough Transform could yield an algorithm that is able to track finer features on the face.

2. Materials, Methods, and Procedures

2.1 Experimental Hardware

My image-processing engine is an ASUS-brand laptop computer with an Intel i7-2830 processor. This CPU runs at 2.2GHz and has 4 cores. Although the laptop has a built-in web camera, I chose to use an external USB camera for more flexibility in set up, specifically in using a more suitable lens to capture the finer features. The camera is monochromatic and made by IDS Imaging, model number UI-122xLE, with a 752x480 resolution. The software development tool I use is the Microsoft Visual Studio C++.

2.2 Methods

2.2.1 Designing the Hough Transform templates

The focus of my project was to use the Hough Transform to detect eyes in facial images. To implement the eye-searching Hough Transform, the first thing I had to do was to develop a set of Hough templates. I used the general eye shapes shown in Figure 1 as a template.

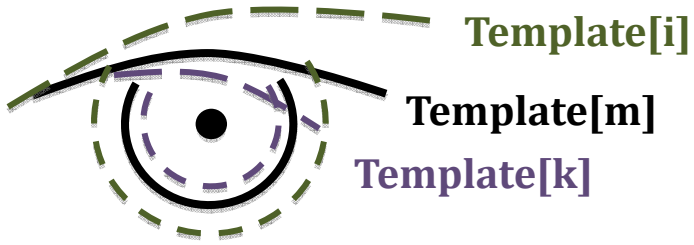

Figure 2. Hough Transform templates used in this project

The set of templates is derived from a basic eyelid, eyeball, and pupil shape by allowing variations in size, spacing, and degree of slant. Overall, there are 7 pupil sizes, 2 eyelid shapes, 5 eyelid slant angles, and 5 eyelid-to-pupil spacing, for a total of 350 templates.

2.2.2 Dealing with the noise-pixel problem

As is the case for many problems, noise is always present and must be planned for. Take, for example, the scenario shown in Figure 2. The eye candidates (shown in red) in both images would receive the same amount of votes, despite the fact that, to a

human observer, the left image clearly does not contain an eye. As a result, the algorithm may end up finding eyes at the wrong locations.

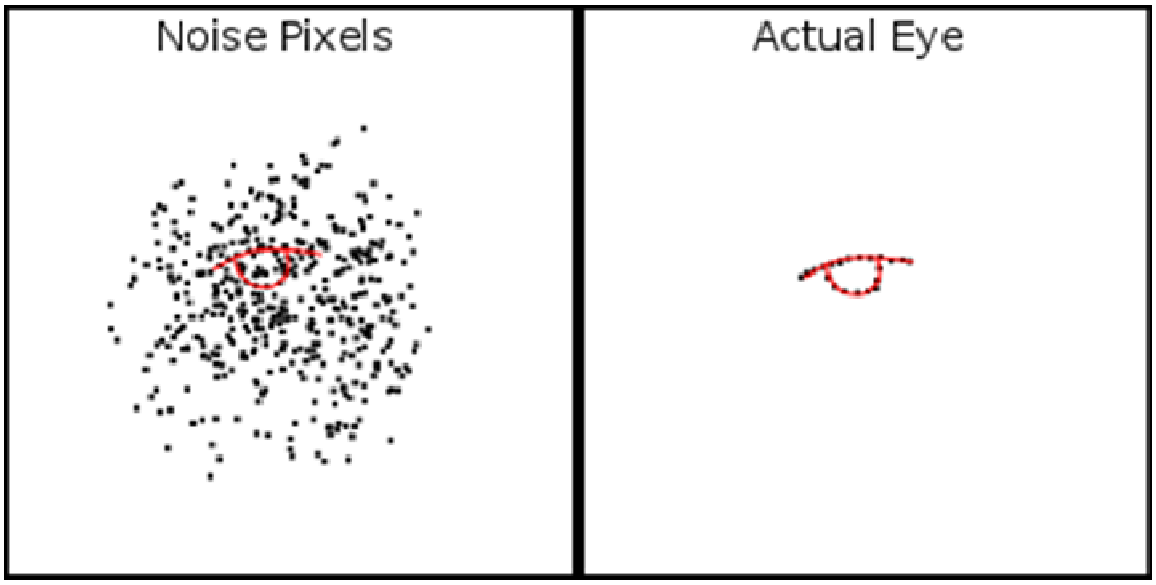

Figure 3. An illustration of the noise problem

To solve this problem, I have included in my algorithm an additional step to reduce the impact of noise pixels. This step is based on the concept that, in Hough Transform (HT) space, peaks corresponding to actual features are more localized, whereas noise pixels produce a more plateau-like response in HT space. This phenomenon is illustrated in Figure 4. In this example the algorithm is searching for a square shape in the image. Image 1 produces a much more pronounced peak in HT space (darker color means more votes) than does image 2. By calculating how pronounced a peak is in HT space, the less pronounced peaks (which likely correspond to patches of noise pixels) in HT space are discriminated against, compared to the more pronounced peaks.

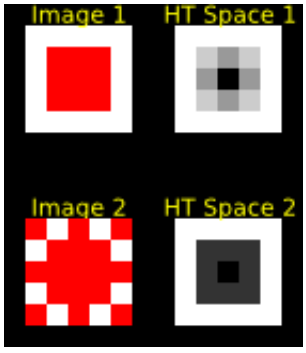

Figure 4. Using HT-space peak profile to reduce noise.

2.2.3 Speed considerations

Although the amount of computation for each Hough Transform is not very large, repeating it for 350 templates, as discussed in section 2.2.1, is demanding. Speed becomes even more important given that I would like my algorithm to approach real-time performance. To make my algorithm faster, I planned to utilize the parallel processing capability of the Intel i7 processor used in my experiment. Since this CPU has 4 cores, I could potentially get a 4x improvement in speed if I parallelize this algorithm. I have obtained the Intel-supplied "TBB" (Thread Building Block) libraries and parallelized the suitable parts of the Hough Transform code. A copy of my parallel HT code is attached at the appendix for reference.

2.3 Procedures

2.3.1 Flow Chart

Figure 5 (below) gives a rough overview of the flow of my algorithm. "Set Camera Live, Binning, and Auto Exposure Control" refers to a one time adjusting of some settings of the camera at the launch of the program; the rest of the steps are all executed each time a new image is received.

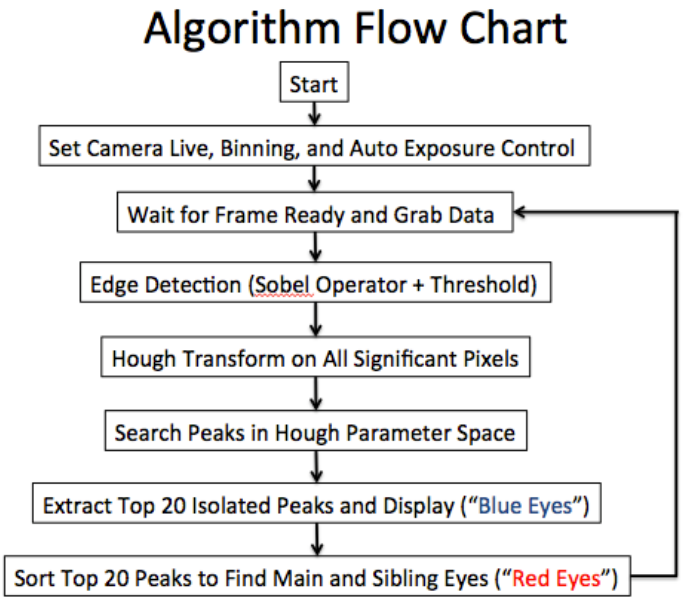

Figure 5. Flow chart of my algorithm

2.3.2 Pre-Processing

The biggest task that I had to complete during pre-processing was edge detection. The Hough Transform works better when the image is composed of sparsely-populated points and the HT uses those points to detect shapes. Thus, the gray-scale image should be transformed into a black-and-white image consisting of only the edges in the original image.

What really determines what is and isn't an edge, though? Ultimately, it comes down to how *quickly* the image changes intensity across a certain pixel. A clearly defined edge would separate a very dark region and a very light region, while a blurry edge would divide two regions with less contrast. I chose to use the Sobel Operator as the basis of my edge-detection algorithm. The Sobel Operator essentially uses mathematics to approximate the gradient of the image intensity at each pixel, and the magnitude of that gradient can be used to determine how well the pixel represents an edge. After calculating the gradient of the image, the magnitude of the gradient at each pixel is compared to a threshold, determined experimentally to allow the process to work under a variety of lighting conditions. Pixels with high enough gradients are deemed edges and the rest are made background pixels. We call these bright edge pixels "significant pixels".

After the first step of pre-processing, the templates to be used with the Hough Transform are generated. These templates would describe varying eye shapes and sizes so that, when the time comes to run the Hough Transform, my algorithm would know what to look for. I came up with 4 parameters for the templates: pupil size, eyelid shape, eyelid slant, and eyelid position. For the sake of speed, I didn't want to have too many templates, but I still needed enough to cover a good portion of the vast variations in human eye shape. Finally, with 7 different pupil sizes, 2 different eyelid shapes, 5 different eyelid slants, and 5 different eyelid positions, the total number of templates came out to be 350. Figure 1 showed an example of some templates. Code for generating the templates is provided in the Appendix (Section 5.1).

2.3.3 The Hough Transform

After Pre-Processing came the Hough Transform, the biggest part of my algorithm. Originally, the Hough Transform was a specific method to find lines in images. I had to expand upon this to utilize it for my purpose, so I came up with a more "generalized" Hough Transform. The whole idea of the generalized Hough Transform is to use each significant pixel in the image to vote for a set of candidate locations for whatever shape the algorithm is searching for. The output of the Hough Transform of an image (using one template) is a two dimensional array with the same size as the image. The intensity of each point on the HT output space is the accumulated votes received for that location, and it represents the likelihood of having the template shape at that location of the image.

To find the set of eye candidates for a given template, all my algorithm needed to do was to loop through all significant pixels in the image and, within that loop, run through all points in the Hough template. Given a certain significant pixel at (x,y) in the image and a point in the Hough template with coordinates (a, b) relative to the template center, the algorithm would let this pixel cast a "vote" for a shape candidate centered at (x-a, y-b). The voting data is stored in a 2D array generally called the accumulator array. My code for the parallelized Hough Transform is provided in the Appendix (Section 5.2) as a reference.

2.3.4 Analyzing the Accumulator

Within the accumulator, the points of greatest interest to me are the ones with the highest number of votes. However, one problem quickly arises: dense patches of noise pixels would receive just as many votes as a real eye (see figure 3 and the discussions therein)! As discussed in 2.2.2, I implemented a way to weigh each HT peak in the accumulator array based on how pronounced the peak is. This weighing process helps to reduce the effect of noise.

At this point, my algorithm selects the top 20 eye candidates out of all the Hough Transform peaks, regardless of template, based on number of votes in Hough Transform and how pronounced the peak is. The candidates not selected are discarded.

After selecting the final 20 candidates, my algorithm tests them for several more attributes that real eyes would have. The most obvious attribute is that eyes come in pairs (I assumed that an entire face is captured in the image), so my algorithm checks whether each candidate had a "sibling" candidate in a location where a counterpart eye might reasonably appear. After this process, the top two eye candidates are displayed on the image in red, and all the remaining eye candidates are displayed in blue. I should note here that the top 20 "blue eyes" are displayed as a way to visualize some of the inner workings of my algorithm, which may help future improvements. The top two candidates ("red eyes") are considered the pair of eyes identified by my algorithm.

3.Results

To test my algorithm, I used a publicly available facial image database from BioID AG. The images can be downloaded from: http://www.bioid.com/download-center/images.html. On average, one run of my algorithm took about 100 milliseconds, which is quite close to real-time performance.

There are 1520 images in the database of various people with different expressions. After running through the entire database blindly, my algorithm correctly identified 65% of the eyes. However, since the database was developed and intended for

use with face detection and not for finer features detection, many of the faces in the images had closed, obscured, or winking eyes, which substantially lowered the identification rate of my algorithm. Nevertheless, I felt this difficult test produced encouraging results.

When the images are appropriate, the algorithm's accuracy is quite good: the correctly identified eyes are precisely matched to the real eyes in the image. Figures 6-9 below show four examples of successful eye detections. Blue eyes indicate the top 20 candidates as described before, and the red eyes indicate the eyes identified by algorithm. As noted above, the blue eyes are for developmental purposes only; they may help suggest ways for future improvement.

Figure 6

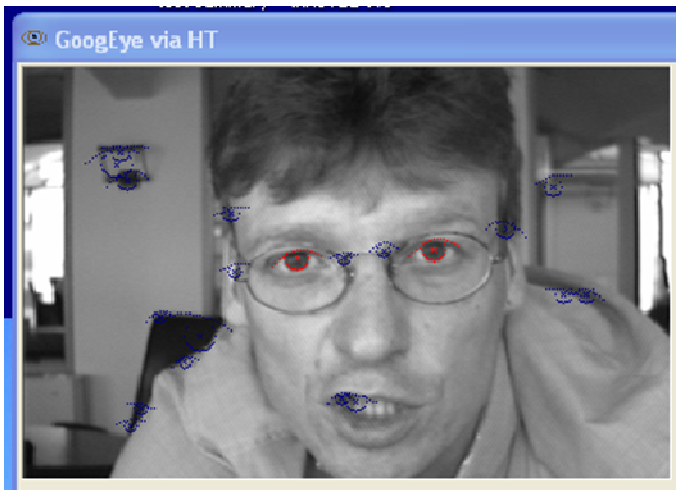

Figure 7

Figure 8

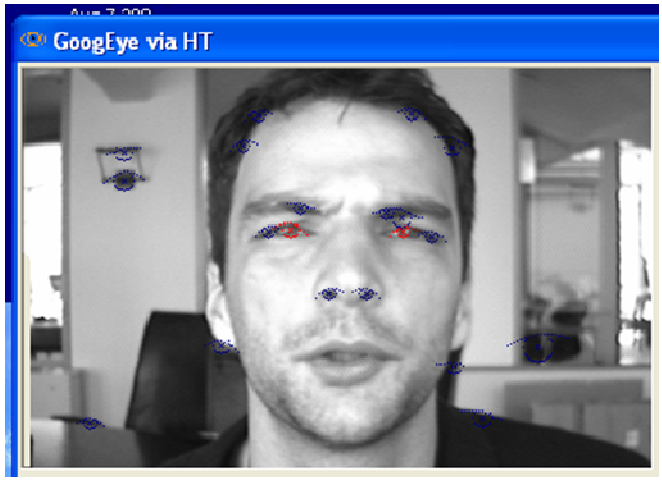

Figure 9

Figures 10-13 below show four examples of failed eye detections:

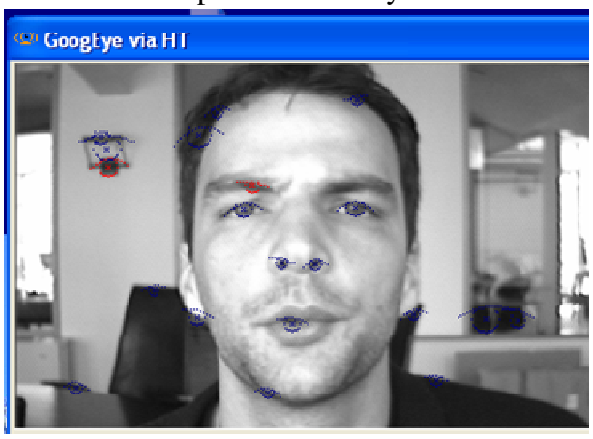

Figure 10

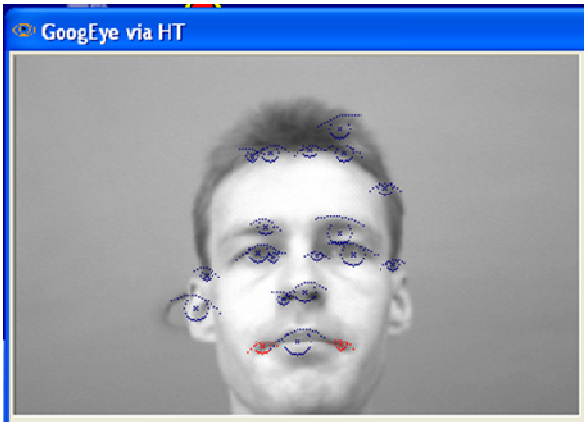

Figure 11

Figure 12

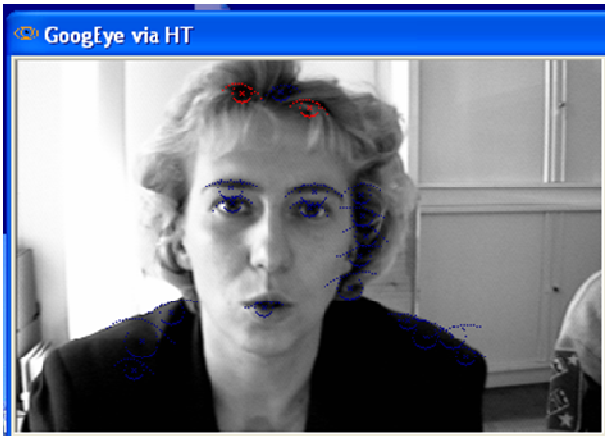

Figure 13

Note that in almost all of the failures, the actual eyes were identified as one of the blue eyes (top 20 candidates).

4. Conclusions and Discussion

4.1 Conclusions

My purpose of doing this experiment is to try to detect and track a fine facial feature using the Hough Transform with everyday computer hardware. After developing and implementing my algorithm, the results look promising and seem to indicate that the Hough Transform is a viable option for fine feature detection for the human face. The different templates used in my algorithm allowed it to not only accurately identify eyes, but also to detect the shapes, sizes, and orientations of the eyes; for example, the algorithm would be able to recognize squinted or widened eyes due to large variations included in the 350 templates. If extended and further improved, this technology could be useful in many areas, such as enabling users to control devices with their eyes or other facial expressions. For example, a device like this can help conserve resources by dimming a TV when the viewer looks away from it, or, in a more significant application, help translating the emotion or needs of a disabled person who can only move his or her facial muscles.

4.2 Failure Analysis

When my algorithm failed, there were several recurring reasons for failure. One of the biggest reasons was obscured or abnormal eyes: the reflections from peoples' glasses would interfere with the edge detection and therefore the entire algorithm, and eyes that were heavily squinted, greatly widened, or otherwise irregular were hard for my algorithm to detect sometimes.

Another major problem that came up was non-ideal conditions in the picture overall. If the lighting was too bright or too dark, there wouldn't be enough contrast in the image for the edge detection to work successfully. In addition, blurry or unfocused images proved to be hard to deal with because the edges in the images weren't clearly defined.

Sometimes, the content of the image was just not ideal. In images with very cluttered backgrounds, many places in the edge-detected version of the image could resemble an eye. If conditions were just right, some object in the background would be detected as the eye while the real eyes were rejected (although they usually made it through the algorithm to the very last stage, into the group of top 20 eyes).

4.3 Future Improvements

One of the biggest similarities between the different causes of failure mentioned in the previous section was that they all showed the lack of robustness in the edge detection portion of my algorithm. I hope to improve on my algorithm in the future to address this problem. I'd like to increase the tolerance of my algorithm to non-ideal

exposure, bad focus, and bad lighting, possibly by making better use of auto-exposure and auto-focus methods.

In addition, I plan to add a component to my algorithm that I believe will not only help increase its efficiency but also its accuracy: delayed image subtraction. Since many applications of this technology will require running the algorithm in real time (and thus placing an emphasis on efficiency and speed of the algorithm), I believe that my algorithm can make use of some of the previously captured images to determine what parts of the image are really useful and what parts are background or stationary. Since the background of the image generally does not change with time, I can determine what parts of the image are background by subtracting two frames taken at different times and analyzing the results. This method could prove immensely useful in taking away many of the false eyes found in the backgrounds of images and at the same time reduce the amount of computation load.

5. References

1); T. Rikert, M. Jones, and P. Viola, “A Cluster-Based Statistical Model for Object Detection,” Proc. Seventh IEEE Int’l Conf. Computer Vision, vol. 2, pp. 1046-1053, 1999.

2); V. Govindaraju, “Locating Human Faces in Photographs,” Int’l J.Computer Vision, vol. 19, no. 2, pp. 129-146, 1996.

3); C. Papageorgiou, M. Oren, and T. Poggio, “A General Framework for Object Detection,” Proc. Sixth IEEE Int’l Conf. Computer Vision, pp. 555-562, 1998.

4); A. Pentland and T. Choudhury, “Face Recognition for Smart Environments,” IEEE Computer, pp. 50-55, 2000.

5); D.H. Ballard, "Generalizing the Hough Transform to Detect Arbitrary Shapes", Pattern Recognition, Vol.13, No.2, p.111-122, 1981

6); W. Burger and M. J. Burge, “Principles of Digital Image Processing”, Springer-Verlag, London, 2009.

7); S. Prata, “C++ Primer Plus”, Fifth Edition, Sams Publishing, Indianapolis 2005.

6. Appendix

6.1 Code for Generating Templates

```
void generateTemplates()
{
	int radius,xrange,i,iLidShape,iLidSlope,iDist,n,Npts;
	float a,b,dummy;
	float slope,dist;  //slope of the eyelid
	for(n=0;n<Ntemplates;n++)  //template index
	{
		radius=n%Npupil+minHoughRadius+PupilRadiusOffset;
		xrange=2*radius;  //eyelid horizontal range is -xrange to +xrange
		iLidShape=(n%(Npupil*Nlidshape))/Npupil;
		b=radius; //a,b are parabola params for eyelid
		if(iLidShape==0)
			a=1.0/(b*4.0); //parabola equation y=-b+a*x^2.
		else
			a=1.0/(b*9.0);
		iLidSlope=(n%(Npupil*Nlidshape*Nlidslope))
				/(Npupil*Nlidshape)-(Nlidslope-1)/2;
		slope=(float)iLidSlope/(2*(Nlidslope-1));  //max slope is +/- 1/4.
		iDist=n/(Npupil*Nlidshape*Nlidslope);
		dist= (iDist-(Ndist-1)/2)*radius/Ndist;  //scale with pupil size
		Hough[n][0].x=0;Hough[n][0].y=0;
		Hough[n][1].x=-1;Hough[n][1].y=-1;
		Hough[n][2].x=-1;Hough[n][2].y=1;
		Hough[n][3].x=1;Hough[n][3].y=-1;
		Hough[n][4].x=1;Hough[n][4].y=1;
		Npts=5;
		for(i=-radius;i<=radius;i++){  // the pupil circle,1/2
			Hough[n][Npts].x=i;
			dummy=radius*radius-i*i;
			Hough[n][Npts].y=(int)( sqrt(dummy) );
			Npts++;
			}
		for(i=-radius;i<=radius;i++){  //  1/4 of the top pupil circle
			if( (i<=-radius*0.75)||(i>=radius*0.75) ){
				Hough[n][Npts].x=i;
				dummy=radius*radius-i*i;
				Hough[n][Npts].y=-(int)( sqrt(dummy) );
				Npts++;
			}
		for(i=-xrange;i<=xrange;i+=2){  //top of parabola
			Hough[n][Npts].x=i;
```

```
                    dummy=-b+a*i*i;
                    Hough[n][Npts].y= dummy+dist+slope*(float)i;
                    Npts++;
                }
                HTsizeX[n]=xrange; HTsizeY[n]=radius+abs(slope)*xrange;
                NpointsHT[n]=Npts;
                HTinc[n] = NmaxHTpoints*1000/Npts;  //increments used in
        //HT. this equalizes the #points difference among templates
        } //next template
}  //*********** END of "generateTemplates()
```

6.2 Code for Parallelized Hough Transform

```
Parallel_for(  blocked_range<int>(0,Ntemplates),HTransf( sigPixel, NsigPixels, Hough,
NpointsHT, HTinc, HTsizeX, HTsizeY, ResultHT )   );

void HTransf::operator()(const blocked_range<int>& r) const
{
        Point* sigpix=sigpix_a;
        int nsigpix=nsigpix_a;
        Point (*Htmpl)[NmaxHTpoints]=Htmpl_a;
        int* NHpoints=NHpoints_a;
        int* incr=incr_a;
        int* sizeX=sizeX_a;
        int* sizeY=sizeY_a;
        int (*result)[imgW][imgH]=result_a;
        int i,j,n,x,y;
        int xbound,ybound;
        for(n=r.begin();n<r.end();n++)
        {
                xbound=sizeX[n]; ybound=sizeY[n];
                for(i=0;i<nsigpix;i++)
                        if( sigpix[i].x>=xbound && sigpix[i].x<(imgW-xbound)&&
                                sigpix[i].y>=ybound && sigpix[i].y<(imgH-ybound)  )  {
                                for(j=0;j<NHpoints[n];j++) {
                                        x=sigpix[i].x-Htmpl[n][j].x;
                                        y=sigpix[i].y-Htmpl[n][j].y;
                                        result[n][x][y] += incr[n];
                                }
                        }
        }
}   //**** END of HTranf() operator def
```