Face Detection for Surveillance

18-551, Spring 2002
Group 2, Final Report:

Avinash Baliga, Dan Bang, Jason Cohen, Carsten Schwicking
{abaliga, hbang, cohen3, carstens} @ andrew.cmu.edu

Introduction

The Big Idea

Face recognition for security purposes is becoming more and more prevalent (especially after recent events). Such systems typically rely on closed circuit television to obtain images. A wireless camera system to obtain images, with a central station to process faces found in images, would be much faster and cheaper to deploy. Each wireless camera would need only a power source, which could be a battery for temporary surveillance or a simple connection to a power grid. Thus, this approach foregoes the need of any major wiring to deploy a system. We intend to maximize the usefulness of such a system by minimizing the bandwidth used by each camera. With each camera using less bandwidth, this system can handle more cameras and/or less optimal transmission conditions.

The Approach

We use a two-stage approach to the problem of face detection. First, we perform segmentation ie. dividing the input image into separate regions of interest. We then apply a detection/verification algorithm, which simply tells us whether each region of interest contains a face or not. We segment by using a skin-tone detector, followed by some light processing to break the image into regions containing skin[1] – this involves morphological filtering and a region labeling (blob-coloring) algorithm. We then use an algorithm devised by Schneiderman & Kanade[2] for actual detection.

---

[1] *Soriano, M.; Martinkauppi, B.; Huovinen, S.; Laaksonen, M.* Skin detection in video under changing illumination conditions. *Pattern Recognition, 2000. Proceedings. 15th International Conference on , Volume: 1* , 2000 Page(s): 839 -842 vol.1

[2] H. Schneiderman and T. Kanade. A Statistical Model for 3D Object Detection Applied to Faces and Cars. *IEEE Conference on Computer Vision and Pattern Recognition*, IEEE, June, 2000

Motivation for Pre-Processing

Schneiderman & Kanade's algorithm takes a lot of computational power to process an entire frame. Schneiderman's algorithm maintains a Boolean map of which pixels could potentially be a face, and each level of evaluation further refines this map. We decided that we could speed up the system by establishing the map's initial values to be true only in regions where skin colors are detected, since a one-pass per-pixel skin transform takes very little time, but could save a lot of time for Schneiderman's algorithm.

Overview of Schneiderman's Algorithm

In order to verify that the output of our preprocessing does contain a face, we chose a face detection algorithm that is able to search extensively across scale and position. Schneiderman & Kanade's algorithm accomplishes this in a reasonable amount of time. In our project, we implemented a subset of his project.

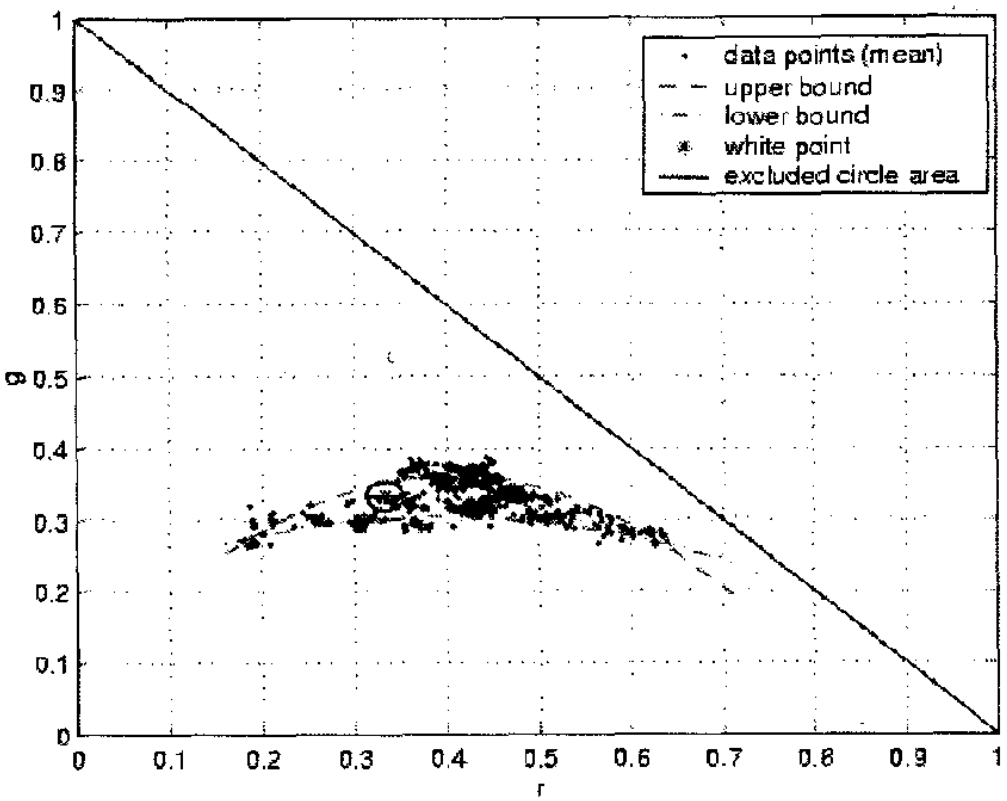
The complete algorithm is composed of

1. wavelet transform:
   A 5/3 linear phase filterbank is used
2. calculation of 17 operators (tests for statistical dependency) based on the wavelet transform coefficients
3. a computation of the probability of the presence of a face

The Algorithms

Pre-Processing – Skin Transform[3]

The skin transform creates a Boolean image which tells whether each pixel in the source image is skin-colored or not. Soriano et. al. took a large sampling of skin colors across different races, and found that skin colors are largely dependent on red-to-green ratios, while lacking relative blue intensity. The spectrum of red-to-green ratios corresponding to skin colors in their test data was fitted to a quadratic equation, so that it would be very computationally inexpensive to decide whether a given color is a skin color. The spectrum of skin colors corresponds to the following chart:



[3] *Soriano, M.; Martinkauppi, B.; Huovinen, S.; Laaksonen, M.* Skin detection in video under changing illumination conditions. *Pattern Recognition, 2000. Proceedings. 15th International Conference on , Volume: 1* , 2000 Page(s): 839 -842 vol.1

Pre-Processing – Morphological Filtering[4]

The output of the skin transform is usually very speckled, since it is a Boolean image rendered from a full color image. Determining where regions of skin appear requires removal of the high frequency content, which in essence removes small speckles and merges high concentration areas into a connected region. Also, since Schneiderman's face detection algorithm only searches for faces at least 20 pixels by 20 pixels, it makes sense to discard skin regions smaller than this size. Morphological filtering seemed like a perfect choice, since the erode operation removes small speckles, and the dilate operation merges nearby regions.

Optimizing Morphological Filtering

Initially we chose a rather inefficient technique of morphological filtering, because we wanted to use it correctly as defined rather than haphazardly. We noted that a close operation (dilate, then erode) connects nearby regions without altering the sizes or positions of any objects, and that an open operation (erode, then dilate) removes small speckles also without altering objects' sizes and positions. We determined through experimentation that using a sequence of alternating close and open operations, with a slightly larger circular structuring element for each, produces excellent results. Even merged regions too small to be useful are filtered out because each open uses a larger element than the preceding close. The last close/open set used a circular structuring element approximately the same size as the smallest faces Schneiderman's algorithm detects. Many non-skin pixels that are falsely accepted by the skin transform are filtered out with this technique.

The problem with this approach is that it consists of a large amount of computation, and it appeared that much of this computation was unnecessary. First, consider the atomic morphological operations occurring when performing close, open, close, open, etc. The actual steps are dilate, erode, erode, dilate, dilate, erode, erode, dilate, etc. We noted that the repeated operations could possibly be merged. The output effect of two dilations with a square structuring element is in fact no different than one dilation where the size of the structuring element is the

[4] Jain, A. Fundamentals of Digital Image Processing. Prentice Hall, 1988.

sum of the original two, minus one. *Note that this is not generally true for all structuring elements, but it is true with squares.*

At first glance, merging these operations may seem much faster, since less repeated looping yields improved cache performance and less time wasted with loop variables and branches. But a closer look at the running time reveals a surprise: Suppose we're filtering an image with *n* pixels. If we are going to dilate by a 3x3 square structuring element, and then again with a 5x5, the time taken $9n + 25n$, which is $34n$. On the other hand, a single pass with a 7x7 produces the same results, but takes $49n$ compared to $34n$. Pushing this even further, dilating with a 3x3 square three times also produces the same results, taking only $9n + 9n + 9n = 27n$ time. The problem is that using a larger structuring element causes many recomputations between nearby pixels, some of which are circumvented when repeatedly using a smaller element and allowing its effect to propagate.

So, what's worse, bad cache performance and lots of loops, or having to compute the same things many times unnecessarily? We attempted to find a way to solve both problems at once. Since larger operations take quadratically longer (linearly increasing edge length = quadratically increasing area), we decided to use the larger structuring element in one pass and cut out as much of the repeated computation as possible. The trick was to break out of inner loops early and to benefit from local similarity.

First, instead of using separate memory chunks for the image and circular structuring element, we chose not to store the structuring element at all, and to simply decide inside the inner loop if we were looking at a pixel that would be in the circle. Since this required more inner loop computations including multiplications, we removed the test and found that a square structuring element produced equally good results with much better performance.

The other important change was to stop the inner loops as soon as the answer was tenable, which cost only one *if* inside the inner loop and cut out more than half of the computation. Since our dilate and erode are completely Boolean – i.e. they do not check if the number of skin pixels in the element is above a threshold, but simply if it is zero or not – the

inner loop rarely needs to run completely to determine the result. Dilate can stop as soon as it finds a one, and erode can stop as soon as it finds a zero. This provides a huge speed-up in areas of diverse ones and zeros. In areas which are mostly the same value, the net result is a push-pull performance balance when alternating between dilations and erosions… the slower one operation is, the faster the other will be. Also, starting with the middle row of the inner loop instead of the top row allows both operations to benefit from spatial redundancy, since the first set of pixels checked are the nearby pixels to the left computed most recently.

We ended up only doing 5 morphological filtering operations (dilate, erode, dilate, erode, dilate) and we achieved similar results to our original 14 step process, with much higher performance. Since all development and refinement of the morphological filters was done in Visual Studio (not on the EVM), we can't provide much information about how much faster each modification made the whole process. We can say that originally we waited a few seconds, and the optimizations made the entire preprocessing step run almost instantly.

Union Find (or Blob-Colouring)[5]

Once we have clearly defined regions of contiguous pixels, following the morphological filtering, we must label each contiguous region with a unique number. This is the final stage of our segmentation, since we can extract sub-images from the overall image after this stage. This algorithm consists of making one pass on the image to causally assign regions to each pixel, then performing a region coalescing phase (which captures cases like tridents), followed by a region-re-labeling pass. The first pass and final pass are each O(N), and the region coalescing phase is O(N^2). However, in practice we have very few regions, and the coalescing is O(N^2) in the number of regions, which usually means it executes in very little time. An extra optimization would have been to implement the union-find (with path-compression) algorithm to bound the time to O(N log*(N)).

[5] Cormen, Leiserson, Rivest. Algorithms. MIT Press, 2001. p. 450.

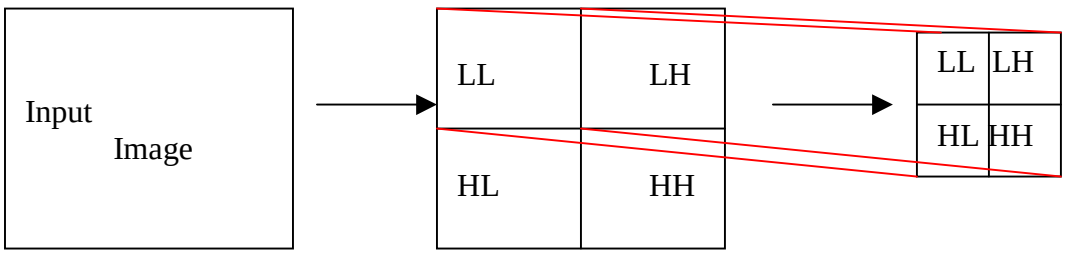
Face Detection As We Implemented It

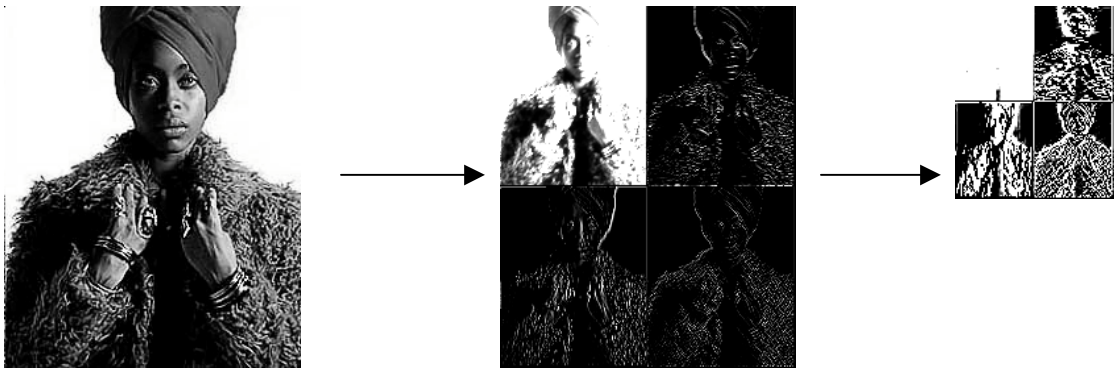
We implemented a subset of Schneiderman & Kanade's algorithm – namely the first evaluator of 17 in their set. The first evaluator is the crudest search, but does most of the work; the remaining 16 evaluators are simply progressive refinements upon the results of the first evaluator. The first evaluator also searches across the entire image, across many scale factors, and is not subject to lighting variations, so it is quite thorough. In brief, the steps involved in the first evaluator are:

- Rescale the Input Image to 4 sizes: the original size (1:1), downsampled by a factor of 1.6818 in each direction, downsampled by a factor of 1.4142 in each direction, and downsampled by a factor of 1.1892 in each direction. This permits search over scale.
- For each of the input images, a 2-Level Dyadic Wavelet Transform
- Principal Component Analysis, which yields a matrix of "feature values"
- Quantization of the feature values
- Probabilistic Analysis, using Training data produced by Schneiderman & Kanade, which yields a matrix of "log likelihood values"
- Thresholding the log likelihood values for each pixel of the image, to determine the likelihood of a face in that pixel's vicinity

Wavelet Transform

The first step in the evaluator is to downsample each of the scalings of the input image by a factor of two both horizontally and vertically, then apply separable low-pass and high-pass filters in all combinations, to yield the four subbands of the wavelet transform: LL, LH, HL, and HH. We call this Level 1. We then take the LL subband, downsample it by a factor of two in each direction, and repeat the process of creating the four subbands. We call this Level 2. This process is illustrated below:





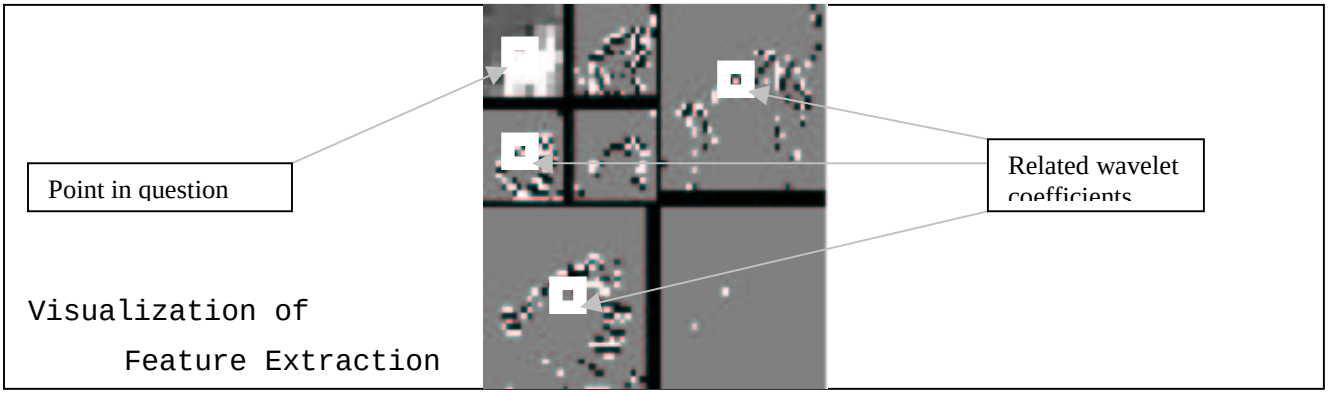
Notice that we do not employ a fast wavelet transform, because we use the LL subband wavelet coefficients in both Level 1 and Level 2.

The wavelet transform is useful in many ways. Like all frequency-based transforms, it concentrates energy in fewer coefficients than the corresponding spatial-domain image pixels, for real-world images. Having more of the energy in fewer coefficients often leads to stronger correlations between coefficients, which can be exploited for detection. Also, intuitively, one can see that highpass-filtering the image is edge-enhancement. Getting an image of just edges (like the LH and HL subbands) can help eliminate the problem of lighting variation, since the main edges of facial features will remain, albeit with some "noise," which in this case would be the edges of shadows.

Principal Component Analysis

Schneiderman & Kanade found statistical dependencies between wavelet coefficients in different subbands and at different levels (1 or 2) when they generated the training data for the first evaluator. In order to limit the computational complexity of the algorithm, they limited the number of dependent coefficients to 8. So, for every wavelet coefficient in LL, Level 2, we make a list of 8 wavelet coefficients that are adjacent to it in the same subband or in different subbands and levels. We then take the inner product of these 8 coefficients and some principal component vectors (that were generated during training), which yields us a (small) list of "principal component coefficients". We then threshold these "PC coefficients" in order to quantize them to one of three values, and sum all the quantized values. This yields a single value we call the "feature value"; since this process is performed for every wavelet coefficient in the

LL subband, Level 2, we end up with a matrix of feature values, which are used in the next step of computation.



Feature matrices are generated for many scalings of the input image, to improve the detection of faces at different scale factors. Specifically, a feature matrix is generated for each of the four scalings of the input image, and every downsampling (by a factor of two) of each of these images, until the image size is smaller than 32x24. In other words, we search across all octaves of scalings, for the 4 originally scaled images, in the hope that we will cover all possible face sizes – and as proven by Schneiderman & Kanade, this yields pretty good results.

The quantization step is performed to reduce computation complexity and memory requirements for the program, since it limits the total number of possible values that could be generated by the algorithm. The feature values are used to index into a precomputed table of probabilities in the next step (generated during training), and the quantization allowed us to store a reasonably sized table.

Probabilistic Analysis – Log Likelihoods

The core idea behind Schneiderman & Kanade's algorithm is Bayesian analysis ie. using conditional probabilities, based on prior knowledge from a training set, to decide whether a group of pixels in an input image correspond to a face. The basic idea is this:

$$\frac{P(image|\ \text{object})}{P(image|\ \text{non-object})} > \lambda$$

Of course, to be able to process such probabilities with high accuracy requires the results of extensive training, for which we have Henry Schneiderman to thank. His training set consisted of nearly 1600 8-bit grayscale images, totaling around 180MB. The images span different races,

ages, genders, lighting conditions, and to a small extent, rotational angles of the faces (for frontal views).

On a side note, it is important to note that many of the calculations involve logarithms simply for computational reasons. P(image | object) involves quite a few products of probabilities, and multiplication is often slower than addition. Logarithms turn products into sums, which are not only faster, but they reduce the range of possible values that follow from a given formula.

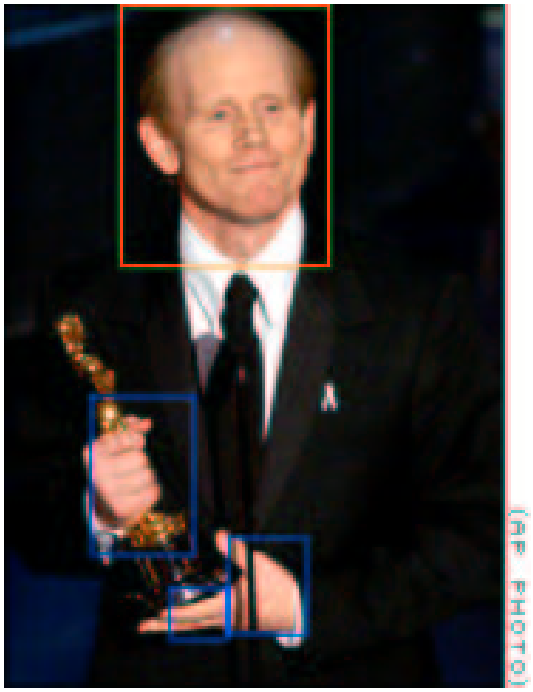
That being said, we can return to the feature matrix. The feature values are used to create what Schneiderman refers to as a "Log-Likelihood Pyramid." Once we have computed all of the feature matrices, we make equivalently sized "log-likelihood matrices" for them. The log-likelihood matrices are computed by taking each feature value from the feature matrix at, say point (M, N), looking it up in a precomputed table (generated during training), and adding the values from the table to the log-likelihood matrix elements centered about (M, N). This is essentially saying, "if the feature value is one we generally associate with faces, increase the probabilities that the pixels in this area are face pixels; if the feature value is not generally associated with faces, then reduce the probability that the surrounding pixels are face pixels."

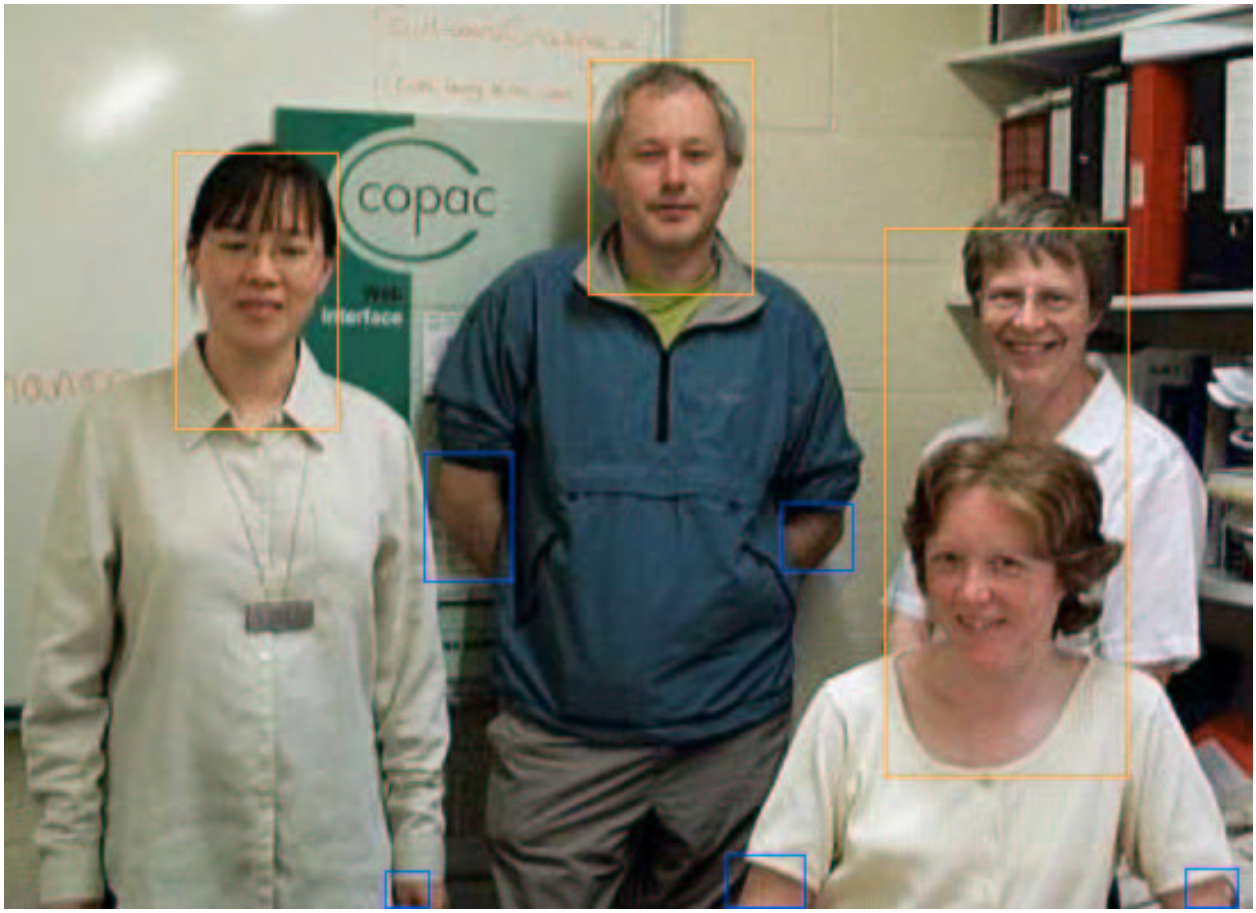
Thresholding

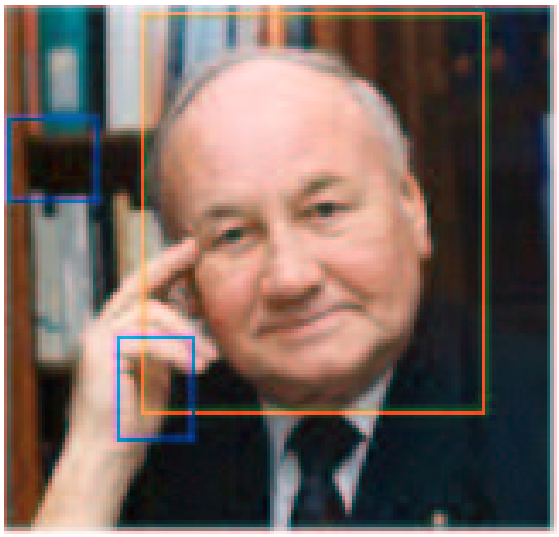
After creating the "Log-Likelihood Pyramid," we have to decide whether there are faces in the picture, and where. As it turns out, we simply look at every element in every log-likelihood matrix, and if it is above a certain threshold (call it lambda), we consider the corresponding pixel in the input image to be the center of a face. Schneiderman decided the value of lambda based on his training data (he ended up with 12). The "log-likelihood" value at each point is a confidence level, so to speak, so once we have a list of points that pass the threshold, we can decide which points are the most likely to be faces as well.

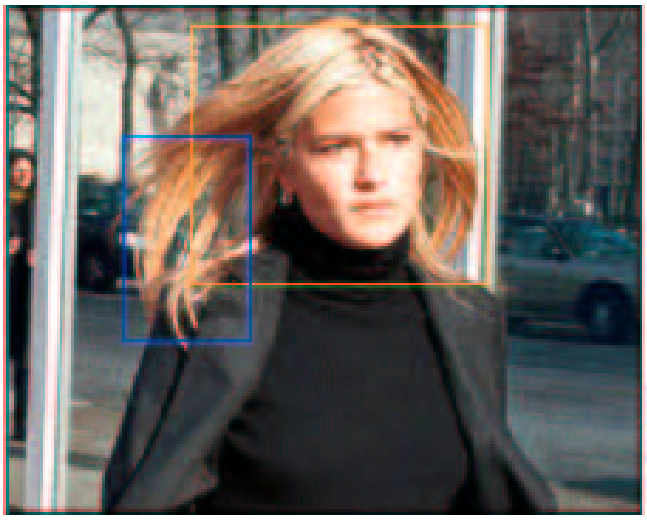
Unfortunately, our implementation of Schneiderman's algorithm did not yield the exact result as Schneiderman's own code. Whereas his code generated a high likelihood centered around a face, we found that our code simply found more points with high confidence level in an image that contained a face or faces. This was still useful enough to produce a face detector, because we had small sub-images (the regions following the morphological filtering) which were generated by the preprocessing stages. We ran our detector on each subimage independently,

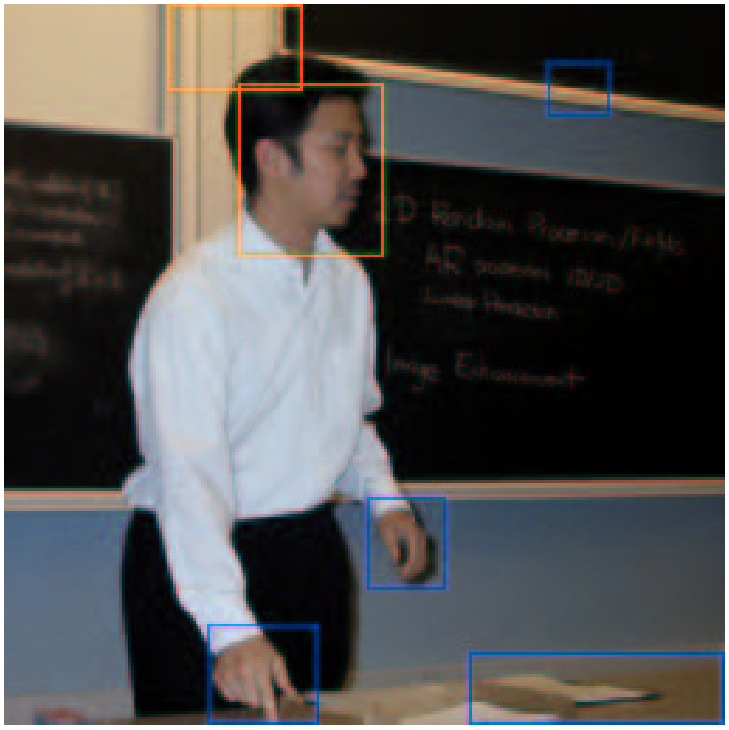
and if the detector found more than three points with high confidence, we deemed the subimage a face. We actually achieved surprisingly good results from this, as evidenced by the following images:




copac




AP PHOTO



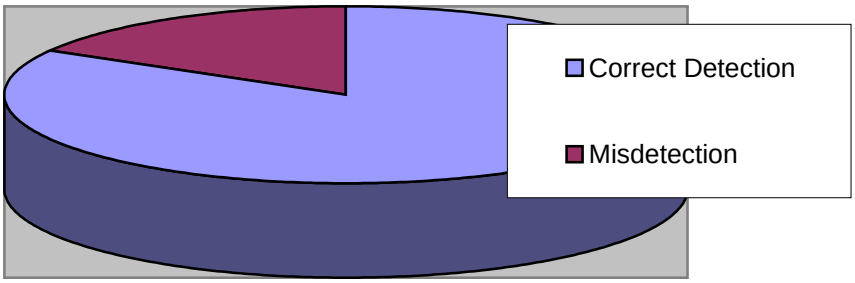
Reliability of Our Detector

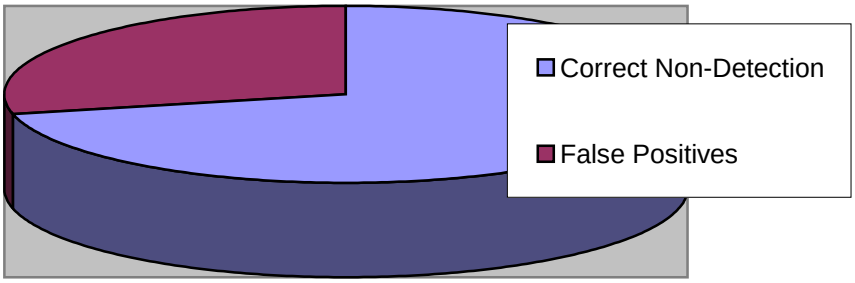
Correct Faces
30 (83.3%)

Correct Non-Faces
25 (71.4%)

False Detections (non-faces detected as faces)
10

Misdetections (faces not detected as faces)
6





Logistics

The PC to EVM Connection

In attempt to make this project feasible, we chose not to attempt to port Schneiderman's algorithm to the EVM. Instead, we perform all the pre-processing on the EVM board, and send a stream of coordinates specifying the regions isolated by the pre-processing back to the PC. We were unable to properly implement the initialization of Schneiderman's potential face location map to the regions found by the pre-processing, so instead we repeatedly call his algorithm on each region.

Despite our choice to leave the Schneiderman algorithm on the PC side, we firmly believe that with enough time and effort this algorithm could run on the EVM alongside our pre-processing. Under Windows NT 4.0, we noticed that our implementation used no more than 2 MB. In order to maintain the spirit of embedded programming, we made sure our pre-processing steps accomplished their goal ---- to ease time and space constraints before starting on the more intensive processing. First, we made sure the individual steps of the pre-processing freed unnecessary structures before moving onto the later steps. This kept the memory usage rather low, and allowed us to discard the color image initially passed in.

Before beginning Schneiderman's algorithm, we made sure we had reduced the allocated memory to a minimum. Besides deallocating all temporary buffers, we discarded all image data outside the regions detected by the pre-processing. The remaining image data needed a way to be passed around and used easily, even though it is discontinuous in the original image. We implemented a structure that allows any number of grayscale images of any size to be stored in a linear region of memory. Each image in the stream can be accessed directly because its width, height, and data are stored contiguously, and placed at an offset within the stream. The beginning of the stream contains an array of these offset values, preceded by the number of offsets and the length of the entire stream. Thus, given nothing more than a pointer to the beginning of the stream, all the images can be copied around easily in a single memcpy or DMA routine, and all data in the images in the stream can be accessed directly. Each image's original

coordinates within the input frame are also stored, in order to properly discern where in the original frame the detected faces are located ---- this facilitates such applications as face tracking, and using multiple images of the same face for improved recognition. The amount of memory saved by discarding the non-skin regions is printed to stdout by our demo program.

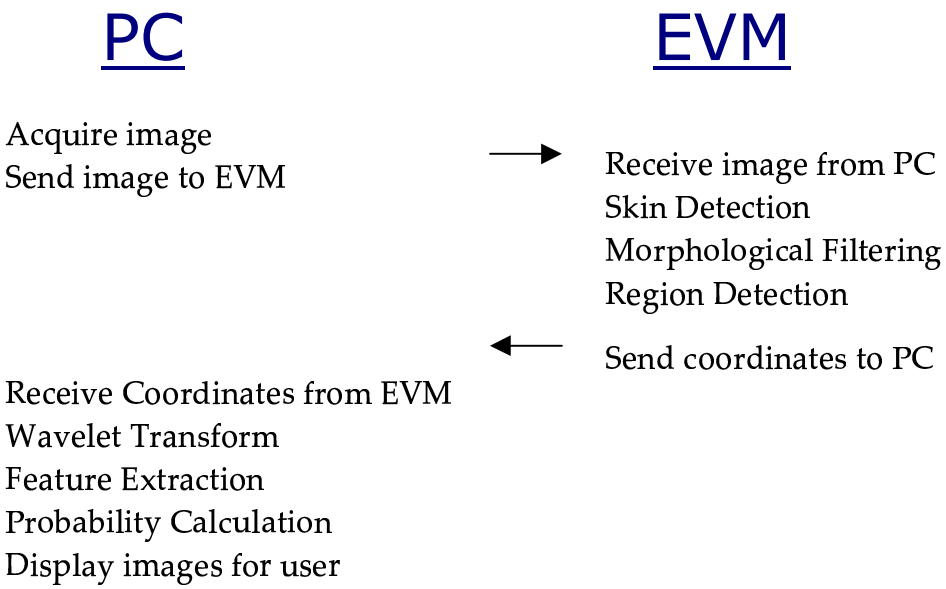
Once the stream of skin-region images arrives on the PC side, we loop through each image and call Schneiderman's algorithm on it. Rather than attempt to discern individual face locations within the skin-region images, we simply used the confidence values to determine whether or not to keep or discard each region, as mentioned earlier. The discarded region images are removed from the stream, and the final compressed stream's actual space requirement, compared with the original image data size, is printed to stdout by our demo. We also output a modified version of the input image with boxes drawn around all the skin-detected regions. The boxes are colored orange if they were accepted by Schneiderman's algorithm, or colored blue if they were rejected.

Image Formats

Our project uses one format in order to work efficiently from preprocessing to verification to output. The input format is portable pixmap (PPM), which simply describes an image through contiguous red, green, and blue channels. This provides easy access for our skin detection point operation, which employs ratios of normalized red and green values.

Because our detection algorithm uses only greyscale inputs, we send only the green channel of the sub-images to be verified. Finally, our output is the input image with bounding rectangles signifying kept sub-images overlaid onto input PPM file.

FlowGraph of Data between PC and EVM



C67 Compiler Output:

```
OUTPUT FILE NAME:   <C:/WINNT/Profiles/551/Desktop/group2_evm/filter.out>
ENTRY POINT SYMBOL: "_c_int00"  address: 000066a0
```

MEMORY CONFIGURATION

name	origin	length	used	attr
ONCHIP_PROG	00000000	00010000	000067e0	R X
SBSRAM_PROG	00400000	00014000	00000000	R X
SBSRAM_DATA	00414000	0002c000	00000000	RW
SDRAM0	02000000	00400000	00400000	RW
SDRAM1	03000000	00400000	000002d0	RW
ONCHIP_DATA	80000000	00010000	0000069c	RW

SBSRAM_PROG is listed as not used but is actually used by our program. We used the hard coded memory addresses to directly write/read from SBSRAM_PROG. This memory is used to store temporary data in our morphological filtering code.

Dynamic data is allocated using malloc (ie on the 4 MB heap on SDRAM0). The EVM proved unreliable in mallocing anywhere near the slightly less than 4 megs of heap that should be available. It would often return null when we attempted to process large images 512x512 and up. The EVM may not be resetting its memory allocation structures properly when being reset, leaving insufficient “free” memory for malloc to work properly.

Results of Optimizing for the EVM

We only have pre-processing running on the EVM, and the skin-transform is very quick compared to the morphological filtering, so we tackled the bottleneck (morphological filtering). The following table summarizes the results of our progressive improvements:

- Circular structuring element, unoptimized:

  overflow in cycle counter

- Circular structuring element, square-skip, (dilate-erode, erode-dilate) to dilate-erode:

  673 Mcycles

- Square structuring element, optimized:

  563 Mcycles

- Virtual square structuring element, optimized & inlined:

  547 Mcycles

- Virtual square structuring element, SBSRAM, DMA transfers:

  527 Mcycles