Summer Research Final Report

Project Title:

Image Processing

Name:

Hassana Ozigi-Otaru

Mentor:

Dr Hong Man

Department of Defense Telehealth Research Experience

Stevens Institute of Technology

May – August 2005

Table of Contents

Table of contents 2-3

Summary 4

Reference Paper on Images and Color Palettes 5-13

Computer Vision, Open CV Library 14-28

Why is Face Detection Important? 29

Reference Paper on Face Detection Techniques 30-38

What is Digital Image Processing? 39

History of Digital Image Processing 40

Significance of Digital Image Processing 41

Edge Detection

Dictionary 45

Appendix: Weekly Reports

Week One

Week Two

Week Three

Week Four

Week Five

Week Six

Week Seven

Week Eight

Week Nine

Week Ten ____

Table of Contents… Cont.d

References ____

Workshop Presentation on Image Processing and Feature Detection

Dr Man’s Workshop Presentation on Image Processing

SUMMARY

This summer I conducted vast research on Image Processing. I started out learning about Face Detection, the algorithms behind it and its significance in society.

I researched detecting other features such as edges based on different limitations, the most common methods for edge detection and the importance of edge detection for the processing of images.

Discovering the world of Image Processing emphasized how significant it is for the advancement of TELEHEALTH. Medical Imaging has evolved over the last few decades and it will keep advancing as long as the technology behind Image Processing is improved.

I came across various Open Source Codes which were a great help, and learnt a whole new vocabulary which I have documented in this report.

NOTE :- THIS REFERENCE ON IMAGES AND PALETTES HAS BEEN DIRECTLY EXTRACTED FROM THE WEBSITE BELOW.

http://hdf.ncsa.uiuc.edu/hdf-java-html/hdfview/UsersGuide/ug06imageview.html#ug06histogram

Chapter 6: Image Viewer

Image Viewer is a graphical window to display HDF images. HDFView is a simple image viewer for HDF4/5 and has very limited function of processing image.

An HDF4 image is raster image of 8-bit pixels with and indexed RGB color table, or a 24-bit true color image. HDF4 library provides image APIs to access image data and color table.

An HDF5 image is a dataset that confirms the HDF5 Image Specification. HDFView supports two types of images: indexed and true color. Both indexed image and true color image have predefined attributes and data layout according to the HDF5 image specification. For more details about HDF5 image, see the HDF5 Image Specification.

- 6.1 Display a 2D or 3D Image
- 6.2 Zoom/Flip/Contour Image
- 6.3 View and Modify Image Palette/Values
- 6.4 Show Histogram of Pixel Values
- 6.5 Import JPEG Image to HDF4/5
- 6.6 Save HDF Image to JPEG File

6.1 Display a 2D or 3D Image

HDFView displays HDF4 raster image or HDF5 datasets that follow the HDF5 Image and Palette Specification for indexed images with a 8-bit standard RGB color model palette or three-dimensional true color images. Other image formats supported by the Image and Palette Specification are not supported by this tool.

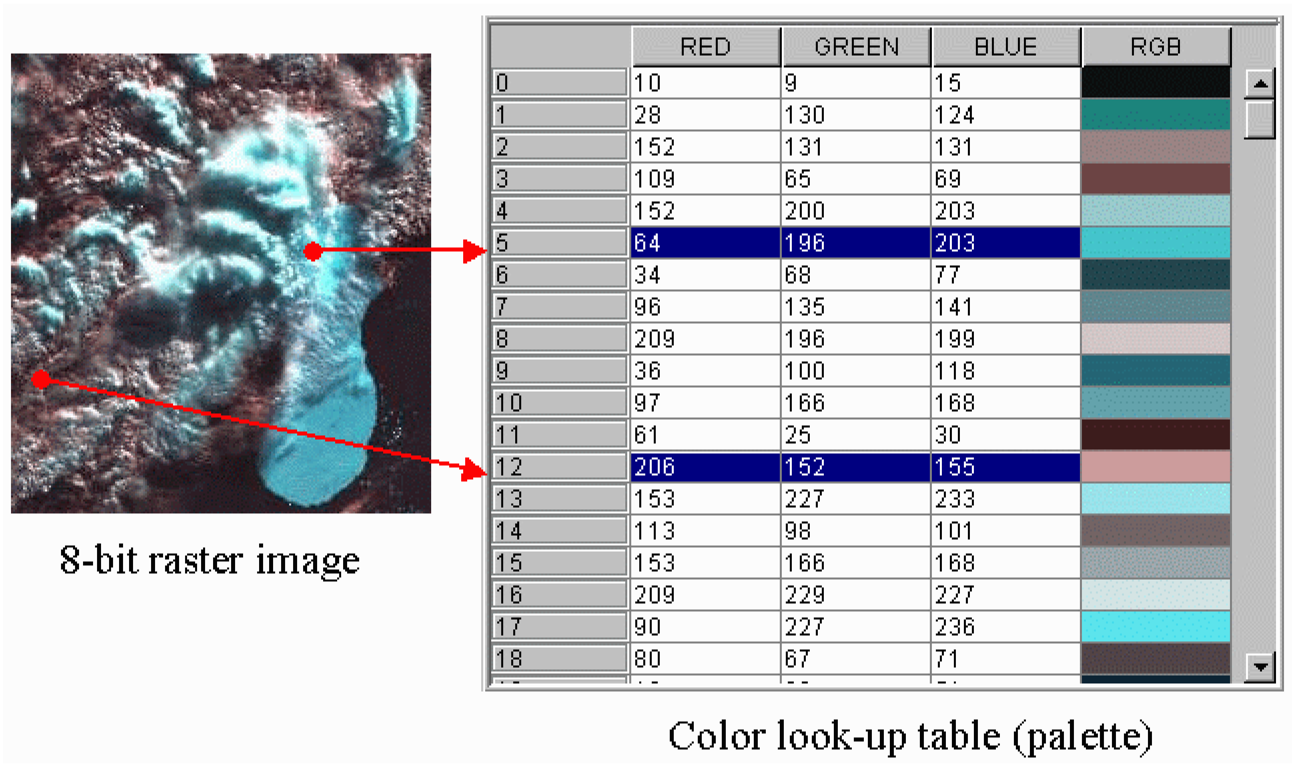
6.1.1 Indexed Image (8-bit)

An indexed image is one of the following:

- An HDF4 RI8 image
- An HDF5 dataset that conforms to the HDF5 Image specification, and is a "IMAGE_SUBCLASS=IMAGE_INDEXED"
- An SDS or HDF5 dataset with data that can be interpreted as an image

The dataset is displayed as a Java image using IndexColorModel. The dataset is converted to a raster image using the first palette specified by the PALETTE attribute, or the default palette for HDF4. Multiple user defined palettes (i.e., the PALETTE attribute may be a list) are not supported in version 1.0.

The dataset of an indexed image holds the values of indices of the color lookup table (palette). The dataset is converted into image pixels by looking up the color table. The following figure is an example of mapping dataset values into pixels.



	RED	GREEN	BLUE	RGB
0	10	9	15	■
1	28	130	124	■
2	152	131	131	■
3	109	65	69	■
4	152	200	203	■
5	64	196	203	■
6	34	68	77	■
7	96	135	141	■
8	209	196	199	■
9	36	100	118	■
10	97	166	168	■
11	61	25	30	■
12	206	152	155	■
13	153	227	233	■
14	113	98	101	■
15	153	166	168	■
16	209	229	227	■
17	90	227	236	■
18	80	67	71	■

Mapping of Dataset Values to Image Pixels

For a two dimensional indexed image HDFView assumes that the width of the image is the size of second dimension and the height of the image is the first dimension, i.e. dim[0]=height and dim[1]=width.

Although HDFView displays the entire image by the order of (dim[0], dim[1], dim[2])=(depth, height, width) by default, you can always change the order and select a subset for display as discussed in Chpater 5.

HDFView also displays a three dimensional array as an array of 2D images arranged along the third dimension, i.e. dim[0]=depth, dim[1]=height and dim[2]=width. You can flip back and forth to look at images at different position of the depth dimension. For instance, if the dataset is 20 x 400 x 600 (dim[0]=20, dim[1]=400, and dim[2]=600), HDFView will display it as 20 images each with the size of 600 x 400 (width is 600, height is 400). However, A three-dimension image of [1][height][width] or [height][width][1] is treated as a two-dimension indexed image of [height][width].

A 2D or 3D SDS or HDF5 dataset with integer or float data can be displayed as an indexed image using the "Open As" selection from the Object menu. Since the dataset does not have a palette, a default palette is used. The palette is chosen from the "Select Palette" menu in the "Dataset Selection" window. The predefined palettes include:

- gray
- rainbow
- nature
- wave

The default is "gray", a gray scale.

6.1.2 True Color Image

In the case of an image with more than one component per pixel (e.g., Red, Green, and Blue), the data may be arranged in one of two ways. HDFView only supports three color components: red, green and blue.

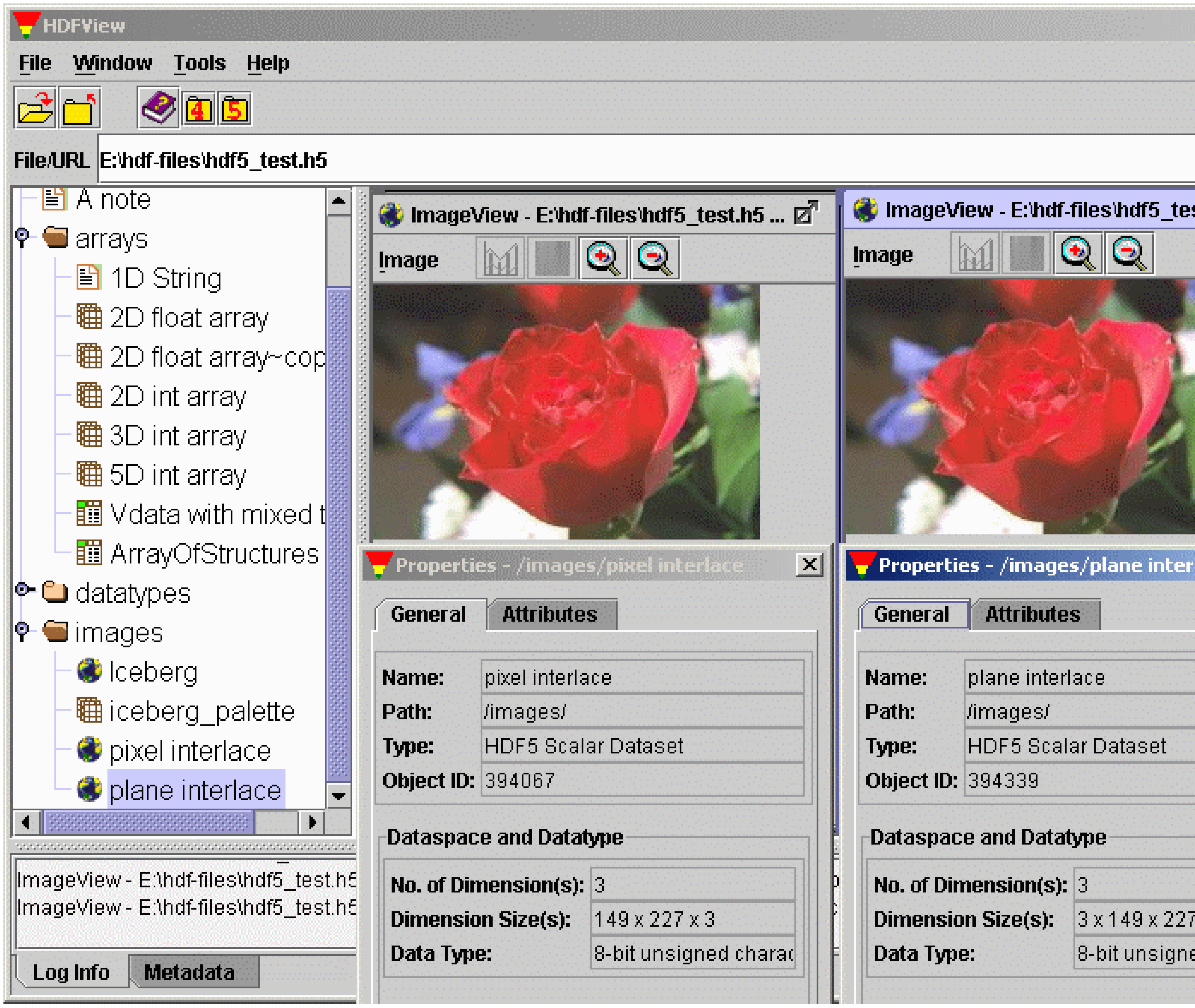
Following HDF4 terminology, the data may be interlaced by *pixel* or by *plane*. For an HDF5 Image dataset the interlace should be indicated by the INTERLACE_MODE attribute. In both cases, the dataset will have a dataspace with three dimensions, *height*, *width*, and *components*. For *pixel interlace* the data is arranged by the order of [height][width][pixel components]. For *plane interlace* the data is arranged by the order [pixel components][height][width].

The translation from pixel values to color components for display or processing purposes is a one-to-one correspondence of data values to components. Data of RGB color components is converted into byte data, which is packed into single *int* pixel. The Java Image is created with a DirectColorModel, with masks to define packed samples. This color model is similar to an X11 TrueColor visual. The default RGB ColorModel specified with the following parameters:

```
Number of bits:        32
Red mask:              0x00ff0000
Green mask:            0x0000ff00
Blue mask:             0x000000ff
Alpha mask:            0xff000000
Color space:           sRGB
isAlphaPremultiplied:  False
Transparency:          Transparency.TRANSLUCENT
```

```
 transferType:          DataBuffer.TYPE_INT
```

The following figure shows examples of true color images. The image on the left is pixel interleaving with dimensions of [149][227][3]. The image on the right is plane interleaving with dimensions of [3][149][227].



File/URL E:\hdf-files\hdf5_test.h5

ImageView - E:\hdf-files\hdf5_test.h5 ...

ImageView - E:\hdf-files\hdf5_tes

ImageView - E:\hdf-files\hdf5_test.h5
ImageView - E:\hdf-files\hdf5_test.h5

Properties - /images/pixel interlace

Name:	pixel interlace
Path:	/images/
Type:	HDF5 Scalar Dataset
Object ID:	394067
No. of Dimension(s):	3
Dimension Size(s):	149 x 227 x 3
Data Type:	8-bit unsigned charac

Properties - /images/plane inter

Name:	plane interlace
Path:	/images/
Type:	HDF5 Scalar Dataset
Object ID:	394339
No. of Dimension(s):	3
Dimension Size(s):	3 x 149 x 227
Data Type:	8-bit unsigne

True Color Image Displayed in the Image View

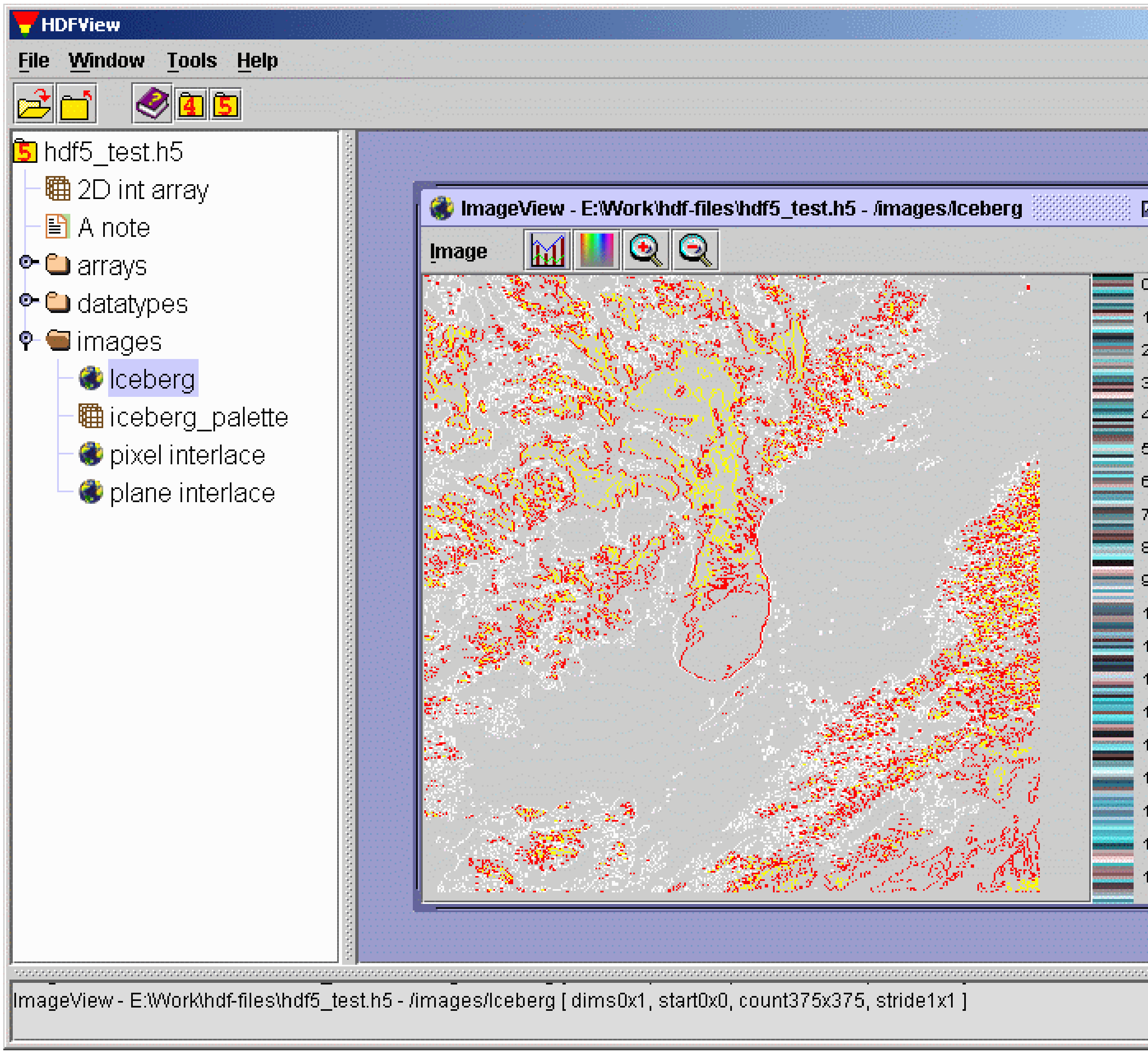
6.2 Zoom/Flip/Contour Image

HDFView supports only limited image manipulation such as zooming, flipping and contour. You can zoom in and out of an image. The minimum zoom factor is 1/8 (reduced to 1/8 the size) and the maximum is 8 (magnified to 8 times the size). Reduction (zoom out) is done by sampling pixels, such as a 1/2 size image is created by selecting every second pixel. Magnification (zoom in) is done by replicating pixels.

You can also flip an image horizontally or verticaly. Flipping an image will change the coordinates of the image. This can be used to adjust images that may have been created with different origins that the defaults.

The "contour" creates a contour plot of the pixel values. The contour can have from three to nine contour levels. Level three has less details of contour and level nine has more details of the contour. **Repeated contour operation shows the accumulated effect of contouring. For example, if you do contouring with level 3 and then do contouring with level 4 on the same image, the final image shows the the effect of contouring with level 4 of the contour image with level 3**.

The following shows an contour image of level nine.



ImageView - E:\Work\hdf-files\hdf5_test.h5 - /images/Iceberg [dims0x1, start0x0, count375x375, stride1x1]

Contour Image

6.3 View and Modify Image Palette/Values

A palette is the means by which color is applied to an image and is also referred to as a color lookup table. It is a table in which every row contains the numerical representation of a particular color. In the example of an 8-bit standard RGB color model palette, this numerical representation of a color is presented as a triplet specifying the intensity of red, green, and blue components that make up each color.

Although the HDF5 palette specification allows for variable color length, different look-up methods and color models beyond RGB, HDFView only supports the indexed RGB color model of 256 colors. Clicking the palette icon from the tool bar or selecting the palette command from the image menu, you can also see the red, green and blue components of the color table are ploted in a line plot.

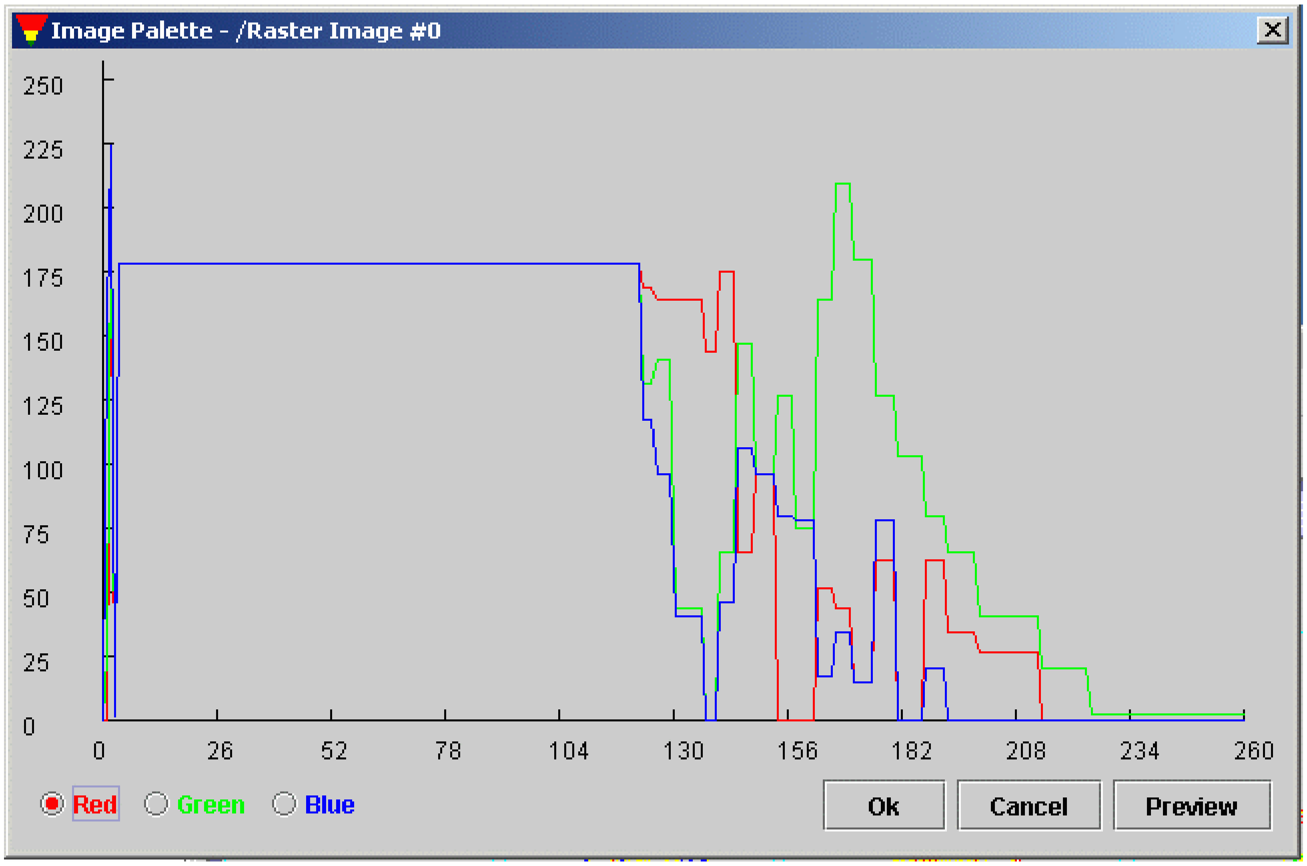
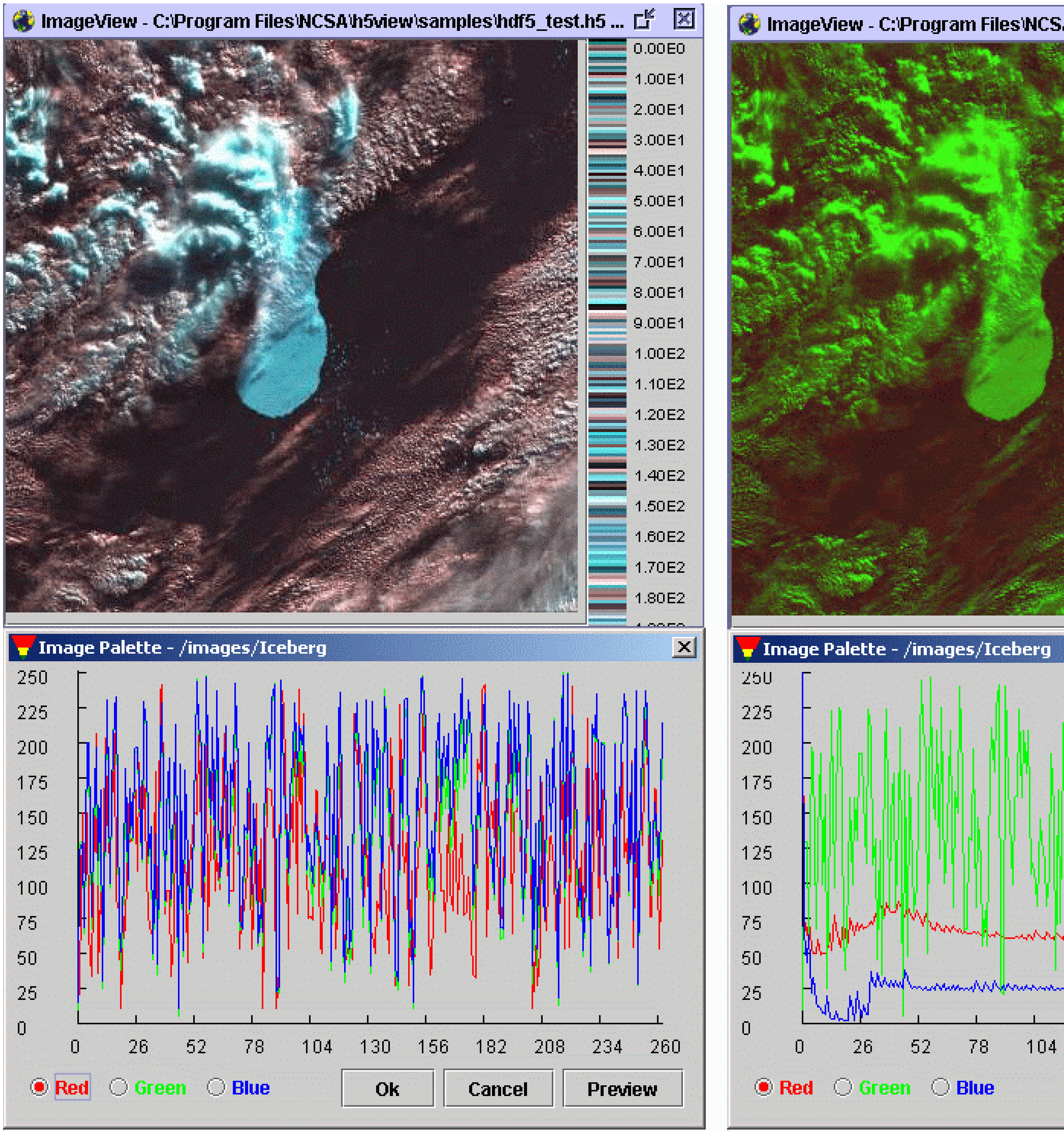


Image Palette (256 Colors)

To view pixel values of each individual point, check the "Show value" item in the "Image" menu. When you move the mouse over the image, the pixel values of the mouse point are shown at the bottom of the image.

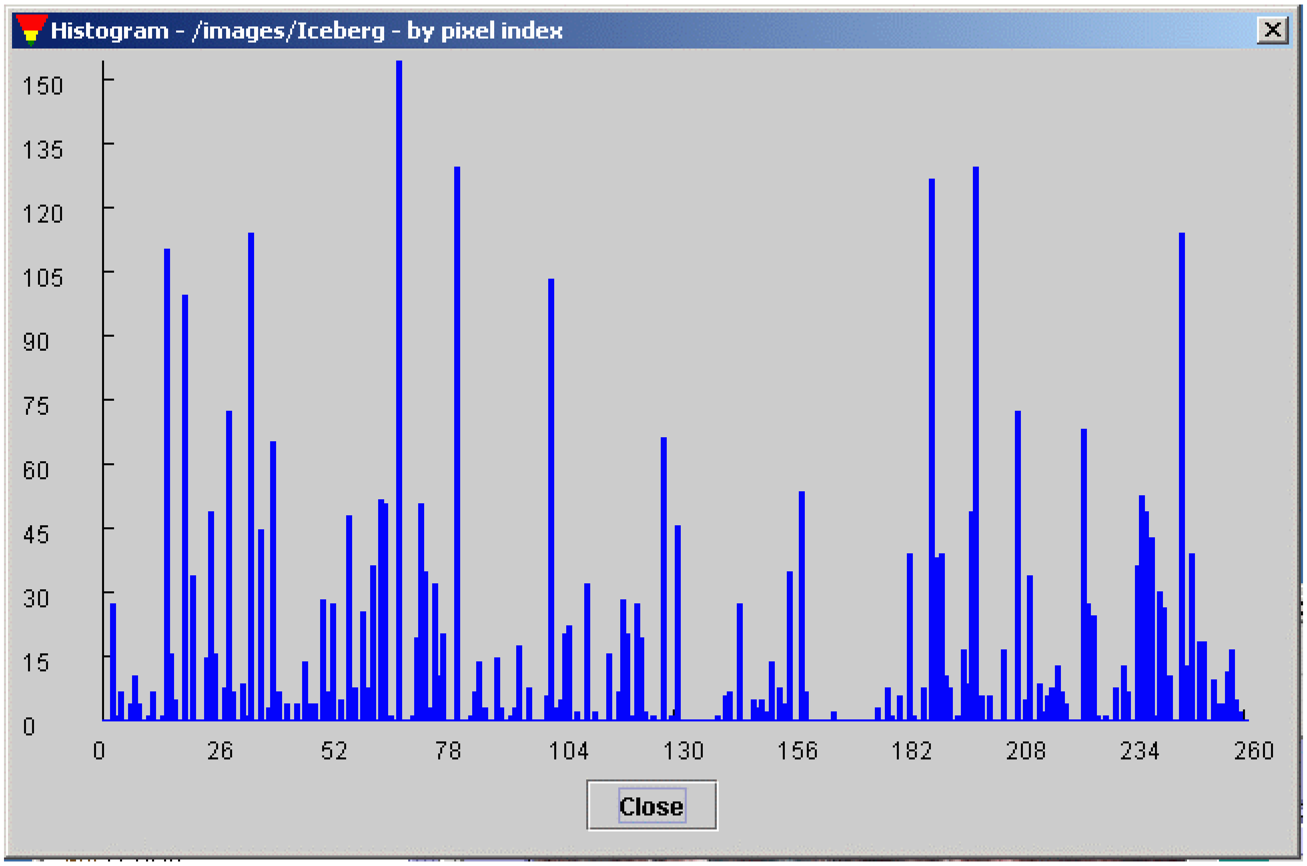
You can modify the values of color table. Select the color (red, green or blue) in the palette view and drag line of selected color. The value of selected color changes as you move the color line. In the following figure, the image on the left is the orginal image and the image on the right is the image with modified color table.



Modified Image Palette (256 Colors)

6.4 Show Histogram of Pixel Values

The frequency of pixel values of a selected area or the whole image can be displayed in a histogram chart. The horizontal axis of the histogram chart is the the 256 pixel values. The vertical axis shows the frequency of the pixel values.



Histogram of Pixel Values

6.5 Import JPEG Image to HDF4/5

Using HDFView, you can convert an JPEG image into HDF4 or HDF5 image. Selec the "Import JPEG To" command in the file menu or the "JPEG To" command in the tools menu, a popup window will ask you to choose an JPEG image file to convert. Image is converted into 24-bit HDF4 or HDF5 image. The current conversion does not support image with indexed color model or image with less than two color components. The image data is saved as 8-bit unsigned integer regardless the data type of the original image.

<u>OPEN CV, OPEN SOURCE COMPUTER VISION LIBRARY</u>

Open CV is a cross platform, middle- to – high API (Application Programming Interface) that consists of hundreds of C functions.

<u>NOTE: THIS REFERENCE HAS BEEN EXTRACTED FROM THE SOURCE BELOW</u>

C:\Program Files\OpenCV\docs\index.htm
C:\Program Files\OpenCV\docs\ref\opencvref_cv.htm

Gradients, Edges and Corners

Sobel

Calculates first, second, third or mixed image derivatives using extended Sobel operator

```
void cvSobel( const CvArr* src, CvArr* dst, int xorder, int yorder, int aperture_size=3 );
```

`src`
: Source image.

`dst`
: Destination image.

`xorder`
: Order of the derivative x .

`yorder`
: Order of the derivative y .

`aperture_size`
: Size of the extended Sobel kernel, must be 1, 3, 5 or 7. In all cases except 1, aperture_size ×aperture_size separable kernel will be used to calculate the derivative. For `aperture_size`=1 3x1 or 1x3 kernel is used (Gaussian smoothing is not done). There is also special value `CV_SCHARR` (=-1) that corresponds to 3x3 Scharr filter that may give more accurate results than 3x3 Sobel. Scharr aperture is:

```
| -3 0  3|
|-10 0 10|
| -3 0  3|
```

for x-derivative or transposed for y-derivative.

The function cvSobel calculates the image derivative by convolving the image with the appropriate kernel:

$$dst(x,y) = d^{xorder+yoder}src/dx^{xorder}\bullet dy^{yorder} \,|_{(x,y)}$$

The Sobel operators combine Gaussian smoothing and differentiation so the result is more or less robust to the noise. Most often, the function is called with (xorder=1, yorder=0, aperture_size=3) or (xorder=0, yorder=1, aperture_size=3) to calculate first x- or y- image derivative. The first case corresponds to

```
  |-1  0  1|
  |-2  0  2|
  |-1  0  1|
```

kernel and the second one corresponds to

```
  |-1 -2 -1|
  | 0  0  0|
  | 1  2  1|
or
  | 1  2  1|
  | 0  0  0|
  |-1 -2 -1|
```

kernel, depending on the image origin (`origin` field of `IplImage` structure). No scaling is done, so the destination image usually has larger by absolute value numbers than the source image. To avoid overflow, the function requires 16-bit destination image if the source image is 8-bit. The result can be converted back to 8-bit using cvConvertScale or cvConvertScaleAbs functions. Besides 8-bit images the function can process 32-bit floating-point images. Both source and destination must be single-channel images of equal size or ROI size.

Laplace

Calculates Laplacian of the image

```
void cvLaplace( const CvArr* src, CvArr* dst, int aperture_size=3 );
```

`src`
: Source image.

`dst`
: Destination image.

`aperture_size`
: Aperture size (it has the same meaning as in cvSobel).

The function cvLaplace calculates Laplacian of the source image by summing second x- and y- derivatives calculated using Sobel operator:

$$dst(x,y) = d^2src/dx^2 + d^2src/dy^2$$

Specifying `aperture_size`=1 gives the fastest variant that is equal to convolving the image with the following kernel:

```
|0  1  0|
|1 -4  1|
|0  1  0|
```

Similar to cvSobel function, no scaling is done and the same combinations of input and output formats are supported.

Canny

Implements Canny algorithm for edge detection

```
void cvCanny( const CvArr* image, CvArr* edges, double threshold1,
              double threshold2, int aperture_size=3 );
```

`image`
: Input image.

`edges`
: Image to store the edges found by the function.

`threshold1`
: The first threshold.

`threshold2`
: The second threshold.

`aperture_size`
: Aperture parameter for Sobel operator (see cvSobel).

The function cvCanny finds the edges on the input image `image` and marks them in the output image `edges` using the Canny algorithm. The smallest of `threshold1` and `threshold2` is used for edge linking, the largest - to find initial segments of strong edges.

PreCornerDetect

Calculates feature map for corner detection

```
void cvPreCornerDetect( const CvArr* image, CvArr* corners, int
aperture_size=3 );
```

`image`
: Input image.

`corners`
: Image to store the corner candidates.

`aperture_size`
: Aperture parameter for Sobel operator (see cvSobel).

The function cvPreCornerDetect calculates the function $D_x^2D_{yy}+D_y^2D_{xx} - 2D_xD_yD_{xy}$ where $D_?$ denotes one of the first image derivatives and $D_{??}$ denotes a second image derivative. The corners can be found as local maximums of the function:

```
// assuming that the image is floating-point
IplImage* corners = cvCloneImage(image);
IplImage* dilated_corners = cvCloneImage(image);
```

```
IplImage* corner_mask = cvCreateImage( cvGetSize(image), 8, 1 );
cvPreCornerDetect( image, corners, 3 );
cvDilate( corners, dilated_corners, 0, 1 );
cvSubS( corners, dilated_corners, corners );
cvCmpS( corners, 0, corner_mask, CV_CMP_GE );
cvReleaseImage( &corners );
cvReleaseImage( &dilated_corners );
```

CornerEigenValsAndVecs

Calculates eigenvalues and eigenvectors of image blocks for corner detection

```
void cvCornerEigenValsAndVecs( const CvArr* image, CvArr* eigenvv,
                               int block_size, int aperture_size=3 );
```

`image`
: Input image.

`eigenvv`
: Image to store the results. It must be 6 times wider than the input image.

`block_size`
: Neighborhood size (see discussion).

`aperture_size`
: Aperture parameter for Sobel operator (see cvSobel).

For every pixel the function cvCornerEigenValsAndVecs considers `block_size` × `block_size` neigborhood S(p). It calcualtes covariation matrix of derivatives over the neigborhood as:

$$M = \begin{vmatrix} \mathrm{sum}_{S(p)}(dI/dx)^2 & \mathrm{sum}_{S(p)}(dI/dx\bullet dI/dy) \\ \mathrm{sum}_{S(p)}(dI/dx\bullet dI/dy) & \mathrm{sum}_{S(p)}(dI/dy)^2 \end{vmatrix}$$

After that it finds eigenvectors and eigenvalues of the matrix and stores them into destination image in form (λ_1, λ_2, x_1, y_1, x_2, y_2), where
λ_1, λ_2 - eigenvalues of `M`; not sorted
(x_1, y_1) - eigenvector corresponding to λ_1
(x_2, y_2) - eigenvector corresponding to λ_2

CornerMinEigenVal

Calculates minimal eigenvalue of gradient matrices for corner detection

```
void cvCornerMinEigenVal( const CvArr* image, CvArr* eigenval, int
block_size, int aperture_size=3 );
```

`image`
: Input image.

`eigenval`

Image to store the minimal eigen values. Should have the same size as `image`

`block_size`

Neighborhood size (see discussion of cvCornerEigenValsAndVecs).

`aperture_size`

Aperture parameter for Sobel operator (see cvSobel). format. In the case of floating-point input format this parameter is the number of the fixed float filter used for differencing.

The function cvCornerMinEigenVal is similar to cvCornerEigenValsAndVecs but it calculates and stores only the minimal eigen value of derivative covariation matrix for every pixel, i.e. min(λ_1, λ_2) in terms of the previous function.

FindCornerSubPix

Refines corner locations

```
void cvFindCornerSubPix( const CvArr* image, CvPoint2D32f* corners,
                         int count, CvSize win, CvSize zero_zone,
                         CvTermCriteria criteria );
```

`image`

Input image.

`corners`

Initial coordinates of the input corners and refined coordinates on output.

`count`

Number of corners.

`win`

Half sizes of the search window. For example, if `win`=(5,5) then 5*2+1 × 5*2+1 = 11 × 11 search window is used.

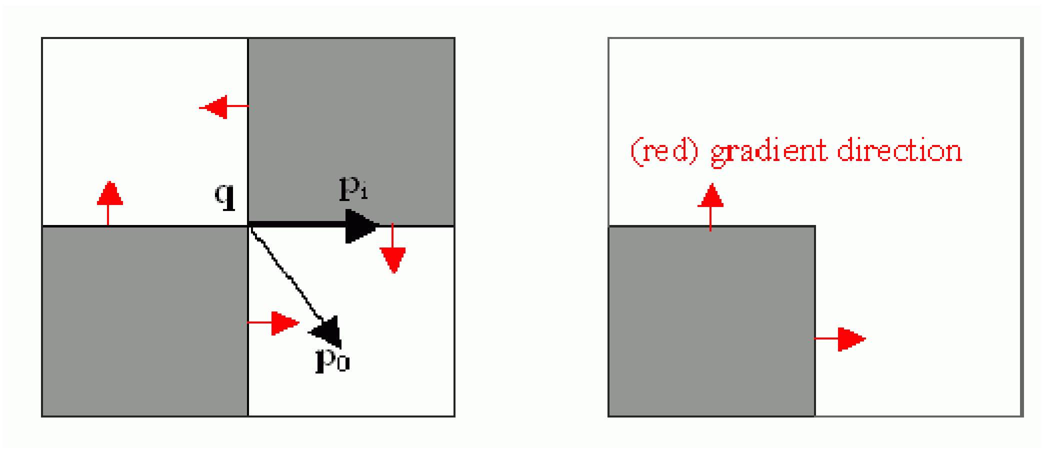
`zero_zone`

Half size of the dead region in the middle of the search zone over which the summation in formulae below is not done. It is used sometimes to avoid possible singularities of the autocorrelation matrix. The value of (-1,-1) indicates that there is no such size.

`criteria`

Criteria for termination of the iterative process of corner refinement. That is, the process of corner position refinement stops either after certain number of iteration or when a required accuracy is achieved. The `criteria` may specify either of or both the maximum number of iteration and the required accuracy.

The function cvFindCornerSubPix iterates to find the sub-pixel accurate location of corners, or radial saddle points, as shown in on the picture below.



Sub-pixel accurate corner locator is based on the observation that every vector from the center `q` to a point `p` located within a neighborhood of `q` is orthogonal to the image gradient at `p` subject to image and measurement noise. Consider the expression:

$$\varepsilon_i = DI_{p_i}^T \bullet (q - p_i)$$

where DI_{p_i} is the image gradient at the one of the points p_i in a neighborhood of `q`. The value of `q` is to be found such that ε_i is minimized. A system of equations may be set up with ε_i' set to zero:

$$\text{sum}_i(DI_{p_i} \bullet DI_{p_i}^T) \bullet q - \text{sum}_i(DI_{p_i} \bullet DI_{p_i}^T \bullet p_i) = 0$$

where the gradients are summed within a neighborhood ("search window") of `q`. Calling the first gradient term `G` and the second gradient term `b` gives:

$$q = G^{-1} \bullet b$$

The algorithm sets the center of the neighborhood window at this new center `q` and then iterates until the center keeps within a set threshold.

GoodFeaturesToTrack

Determines strong corners on image

```
void cvGoodFeaturesToTrack( const CvArr* image, CvArr* eig_image,
CvArr* temp_image,
                            CvPoint2D32f* corners, int* corner_count,
                            double quality_level, double min_distance,
                            const CvArr* mask=NULL );
```

`image`
    The source 8-bit or floating-point 32-bit, single-channel image.
`eig_image`
    Temporary floating-point 32-bit image of the same size as `image`.
`temp_image`
    Another temporary image of the same size and same format as `eig_image`.

`corners`
    Output parameter. Detected corners.
`corner_count`
    Output parameter. Number of detected corners.
`quality_level`
    Multiplier for the maxmin eigenvalue; specifies minimal accepted quality of image corners.
`min_distance`
    Limit, specifying minimum possible distance between returned corners; Euclidian distance is used.
`mask`
    Region of interest. The function selects points either in the specified region or in the whole image if the mask is NULL.

The function cvGoodFeaturesToTrack finds corners with big eigenvalues in the image. The function first calculates the minimal eigenvalue for every source image pixel using cvCornerMinEigenVal function and stores them in `eig_image`. Then it performs non-maxima suppression (only local maxima in 3x3 neighborhood remain). The next step is rejecting the corners with the minimal eigenvalue less than `quality_level`•max(`eig_image`(x,y)). Finally, the function ensures that all the corners found are distanced enough from one another by considering the corners (the most strongest corners are considered first) and checking that the distance between the newly considered feature and the features considered earlier is larger than `min_distance`. So, the function removes the features than are too close to the stronger features.

Filters and Color Conversion

Smooth

Smooths the image in one of several ways

```
void cvSmooth( const CvArr* src, CvArr* dst,
               int smoothtype=CV_GAUSSIAN,
               int param1=3, int param2=0, double param3=0 );
```

`src`
    The source image.
`dst`
    The destination image.
`smoothtype`
    Type of the smoothing:

- CV_BLUR_NO_SCALE (simple blur with no scaling) - summation over a pixel `param1`×`param2` neighborhood. If the neighborhood size may vary, one may precompute integral image with cvIntegral function.
- CV_BLUR (simple blur) - summation over a pixel `param1`×`param2` neighborhood with subsequent scaling by 1/(`param1`•`param2`).
- CV_GAUSSIAN (gaussian blur) - convolving image with `param1`×`param2` Gaussian kernel.
- CV_MEDIAN (median blur) - finding median of `param1`×`param1` neighborhood (i.e. the neighborhood is square).
- CV_BILATERAL (bilateral filter) - applying bilateral 3x3 filtering with color sigma=`param1` and space sigma=`param2`. Information about bilateral filtering can be found at http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

`param1`
: The first parameter of smoothing operation.

`param2`
: The second parameter of smoothing operation. In case of simple scaled/non-scaled and Gaussian blur if `param2` is zero, it is set to `param1`.

`param3`
: In case of Gaussian parameter this parameter may specify Gaussian sigma (standard deviation). If it is zero, it is calculated from the kernel size:

```
                sigma = (n/2 - 1)*0.3 + 0.8, where n=param1 for
horizontal kernel,
                                                    n=param2 for
vertical kernel.
```

Using standard sigma for small kernels (3×3 to 7×7) gives better speed. If `param3` is not zero, while `param1` and `param2` are zeros, the kernel size is calculated from the sigma (to provide accurate enough operation).

The function cvSmooth smooths image using one of several methods. Every of the methods has some features and restrictions listed below

Blur with no scaling works with single-channel images only and supports accumulation of 8-bit to 16-bit format (similar to cvSobel and cvLaplace) and 32-bit floating point to 32-bit floating-point format.

Simple blur and Gaussian blur support 1- or 3-channel, 8-bit and 32-bit floating point images. These two methods can process images in-place.

Median and bilateral filters work with 1- or 3-channel 8-bit images and can not process images in-place.

Filter2D

Convolves the image with the kernel

```
void cvFilter2D( const CvArr* src, CvArr* dst,
                 const CvMat* kernel,
                 CvPoint anchor=cvPoint(-1,-1));
#define cvConvolve2D cvFilter2D
```

`src`
: The source image.

`dst`
: The destination image.

`kernel`
: Convolution kernel, single-channel floating point matrix. If you want to apply different kernels to different channels, split the image using cvSplit into separate color planes and process them individually.

`anchor`
: The anchor of the kernel that indicates the relative position of a filtered point within the kernel. The anchor shoud lie within the kernel. The special default value (-1,-1) means that it is at the kernel center.

The function cvFilter2D applies arbitrary linear filter to the image. In-place operation is supported. When the aperture is partially outside the image, the function interpolates outlier pixel values from the nearest pixels that is inside the image.

Integral

Calculates integral images

```
void cvIntegral( const CvArr* image, CvArr* sum, CvArr* sqsum=NULL,
CvArr* tilted_sum=NULL );
```

`image`
: The source image, `W×H`, single-channel, 8-bit, or floating-point (32f or 64f).

`sum`
: The integral image, `W+1×H+1`, single-channel, 32-bit integer or double precision floating-point (64f).

`sqsum`
: The integral image for squared pixel values, `W+1×H+1`, single-channel, double precision floating-point (64f).

`tilted_sum`
: The integral for the image rotated by 45 degrees, `W+1×H+1`, single-channel, the same data type as `sum`.

The function cvIntegral calculates one or more integral images for the source image as following:

$\texttt{sum(X,Y)=sum}_{x<X,y<Y}\texttt{image(x,y)}$

$\texttt{sqsum(X,Y)=sum}_{x<X,y<Y}\texttt{image(x,y)}^2$

$\texttt{tilted\_sum(X,Y)=sum}_{y<Y,abs(x-X)<y}\texttt{image(x,y)}$

Using these integral images, one may calculate sum, mean, standard deviation over arbitrary pixel up-right or rotated rectangle in O(1), for example:

$\texttt{sum}_{x1<=x<x2,y1<=y<y2}\texttt{image(x,y)=sum(x2,y2)-sum(x1,y2)-sum(x2,y1)+sum(x1,x1)}$

It makes possible to do a fast blurring or fast block correlation with variable window size etc.

CvtColor

Converts image from one color space to another

```
void cvCvtColor( const CvArr* src, CvArr* dst, int code );
```

`src`
    The source 8-bit or floating-point image.
`dst`
    The destination 8-bit or floating-point image.
`code`
    Color conversion operation that can be specifed using CV_<src_color_space>2<dst_color_space> constants (see below).

The function cvCvtColor converts input image from one color space to another. The function ignores `colorModel` and `channelSeq` fields of `IplImage` header, so the source image color space should be specified correctly (including order of the channels in case of RGB space, e.g. BGR means 24-bit format with B_0 G_0 R_0 B_1 G_1 R_1 ... layout, whereas RGB means 24-format with R_0 G_0 B_0 R_1 G_1 B_1 ... layout). The function can do the following transformations:

- Transformations within RGB space like adding/removing alpha channel, reversing the channel order, conversion to/from 16-bit RGB color (Rx5:Gx6:Rx5) color, 15-bit RGB color as well as conversion to/from grayscale using:
- `RGB[A]->Gray: Y=0.212671*R + 0.715160*G + 0.072169*B + 0*A`
- `Gray->RGB[A]: R=Y G=Y B=Y A=0`

  All the possible combinations of input and output format are allowed here.

- RGB<=>XYZ (CV_BGR2XYZ, CV_RGB2XYZ, CV_XYZ2BGR, CV_XYZ2RGB):
- `|X|   |0.412411  0.357585  0.180454| |R|`
- `|Y| = |0.212649  0.715169  0.072182|*|G|`

- `|Z|   |0.019332  0.119195  0.950390| |B|`
-
- `|R|   | 3.240479  -1.53715  -0.498535| |X|`
- `|G| = |-0.969256   1.875991  0.041556|*|Y|`
- `|B|   | 0.055648  -0.204043  1.057311| |Z|`
- RGB<=>YCrCb (CV_BGR2YCrCb, CV_RGB2YCrCb, CV_YCrCb2BGR, CV_YCrCb2RGB)
- `Y=0.299*R + 0.587*G + 0.114*B`
- `Cr=(R-Y)*0.713 + 128`
- `Cb=(B-Y)*0.564 + 128`
-
- `R=Y + 1.403*(Cr - 128)`
- `G=Y - 0.344*(Cr - 128) - 0.714*(Cb - 128)`
- `B=Y + 1.773*(Cb - 128)`
- RGB=>HSV (CV_BGR2HSV,CV_RGB2HSV)
- `V=max(R,G,B)`
- `S=(V-min(R,G,B))*255/V   if V!=0, 0 otherwise`
-
- `       (G - B)*60/S,  if V=R`
- `H= 180+(B - R)*60/S,  if V=G`
- `   240+(R - G)*60/S,  if V=B`
-
- `if H<0 then H=H+360`

  The hue values calcualted using the above formulae vary from 0° to 360° so they are divided by 2 to fit into 8 bits.

- RGB=>Lab (CV_BGR2Lab, CV_RGB2Lab)
- `|X|   |0.433910  0.376220  0.189860| |R/255|`
- `|Y| = |0.212649  0.715169  0.072182|*|G/255|`
- `|Z|   |0.017756  0.109478  0.872915| |B/255|`
-
- L = 116*Y$^{1/3}$ for Y>0.008856
- `L = 903.3*Y      for Y<=0.008856`
-
- `a = 500*(f(X)-f(Y))`
- `b = 200*(f(Y)-f(Z))`
- where f(t)=t$^{1/3}$ for t>0.008856
- `      f(t)=7.787*t+16/116   for t<=0.008856`

  The above formulae have been taken from http://www.cica.indiana.edu/cica/faq/color_spaces/color.spaces.html

- Bayer=>RGB (CV_BayerBG2BGR, CV_BayerGB2BGR, CV_BayerRG2BGR, CV_BayerGR2BGR,
  CV_BayerBG2RGB, CV_BayerRG2BGR, CV_BayerGB2RGB,
  CV_BayerGR2BGR,

CV_BayerRG2RGB, CV_BayerBG2BGR, CV_BayerGR2RGB, CV_BayerGB2BGR)

Bayer pattern is widely used in CCD and CMOS cameras. It allows to get color picture out of a single plane where R,G and B pixels (sensors of a particular component) are interleaved like this:

R	G	R	G	R
G	B	G	B	G
R	G	R	G	R
G	B	G	B	G
R	G	R	G	R
G	B	G	B	G

The output RGB components of a pixel are interpolated from 1, 2 or 4 neighbors of the pixel having the same color. There are several modifications of the above pattern that can be achieved by shifting the pattern one pixel left and/or one pixel up. The two letters C_1 and C_2 in the conversion constants CV_Bayer$C_1C_2$2{BGR|RGB} indicate the particular pattern type - these are components from the second row, second and third columns, respectively. For example, the above pattern has very popular "BG" type.

Threshold

Applies fixed-level threshold to array elements

```
void cvThreshold( const CvArr* src, CvArr* dst, double threshold,
                  double max_value, int threshold_type );
```

`src`
: Source array (single-channel, 8-bit of 32-bit floating point).

`dst`
: Destination array; must be either the same type as `src` or 8-bit.

`threshold`
: Threshold value.

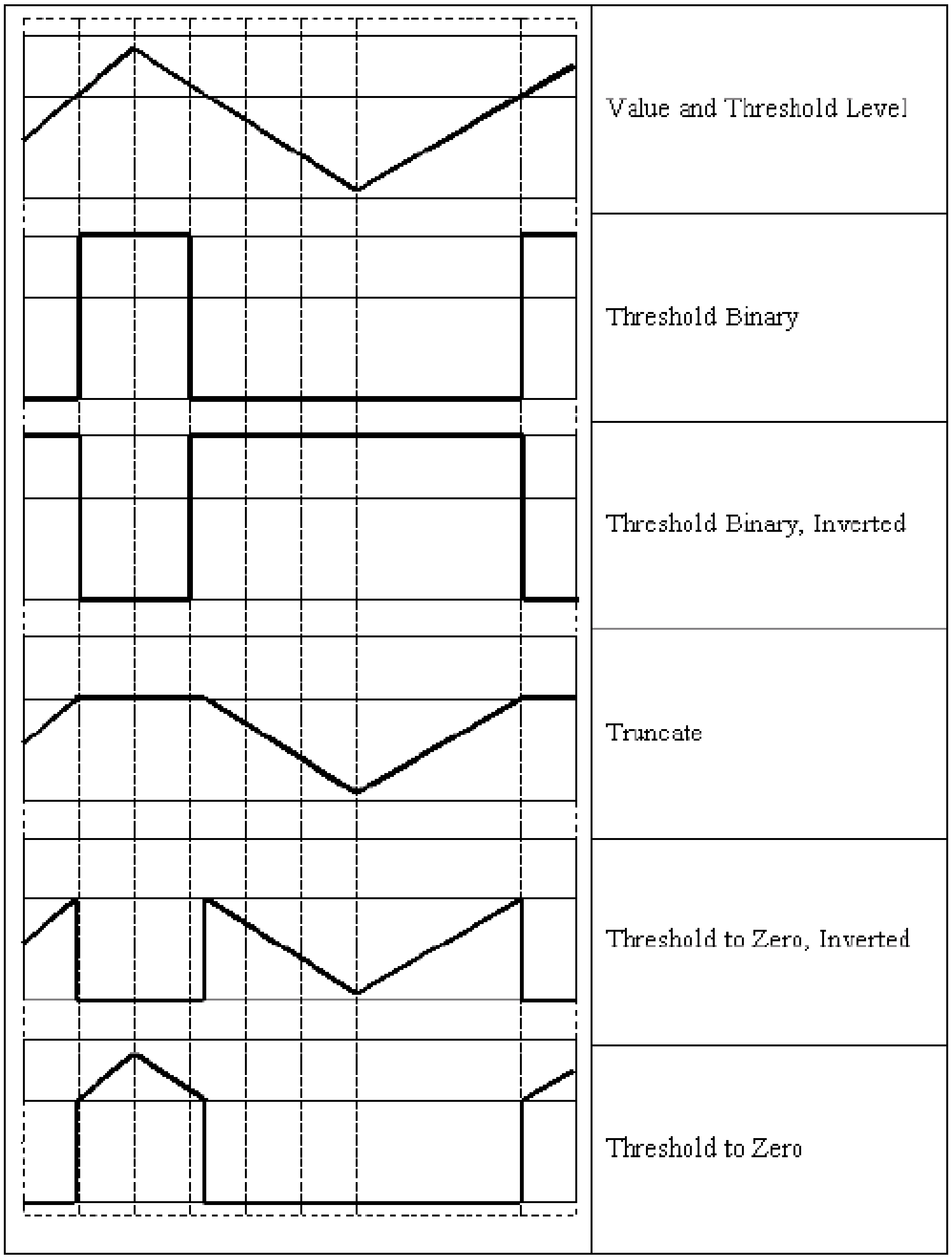
`max_value`
: Maximum value to use with `CV_THRESH_BINARY` and `CV_THRESH_BINARY_INV` thresholding types.

`threshold_type`
: Thresholding type (see the discussion)

The function cvThreshold applies fixed-level thresholding to single-channel array. The function is typically used to get bi-level (binary) image out of grayscale image (cvCmpS could be also used for this purpose) or for removing a noise, i.e. filtering out pixels with too small or too large values. There are several types of thresholding the function supports that are determined by `threshold_type`:

```
threshold_type=CV_THRESH_BINARY:
dst(x,y) = max_value, if src(x,y)>threshold
           0, otherwise

threshold_type=CV_THRESH_BINARY_INV:
dst(x,y) = 0, if src(x,y)>threshold
           max_value, otherwise

threshold_type=CV_THRESH_TRUNC:
dst(x,y) = threshold, if src(x,y)>threshold
           src(x,y), otherwise

threshold_type=CV_THRESH_TOZERO:
dst(x,y) = src(x,y), if (x,y)>threshold
           0, otherwise

threshold_type=CV_THRESH_TOZERO_INV:
dst(x,y) = 0, if src(x,y)>threshold
           src(x,y), otherwise
```

And this is the visual description of thresholding types:


Value and Threshold Level
Threshold Binary
Threshold Binary, Inverted
Truncate
Threshold to Zero, Inverted
Threshold to Zero

AdaptiveThreshold

Applies adaptive threshold to array

```
void cvAdaptiveThreshold( const CvArr* src, CvArr* dst, double
max_value,
                          int
adaptive_method=CV_ADAPTIVE_THRESH_MEAN_C,
                          int threshold_type=CV_THRESH_BINARY,
                          int block_size=3, double param1=5 );
```

`src`
: Source image.

`dst`
: Destination image.

`max_value`
: Maximum value that is used with `CV_THRESH_BINARY` and `CV_THRESH_BINARY_INV`.

`adaptive_method`
: Adaptive thresholding algorithm to use: `CV_ADAPTIVE_THRESH_MEAN_C` or `CV_ADAPTIVE_THRESH_GAUSSIAN_C` (see the discussion).

`threshold_type`
: Thresholding type; must be one of

- `CV_THRESH_BINARY,`
- `CV_THRESH_BINARY_INV`

`block_size`
: The size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, ...

`param1`
: The method-dependent parameter. For the methods `CV_ADAPTIVE_THRESH_MEAN_C` and `CV_ADAPTIVE_THRESH_GAUSSIAN_C` it is a constant subtracted from mean or weighted mean (see the discussion), though it may be negative.

The function cvAdaptiveThreshold transforms grayscale image to binary image according to the formulae:

```
threshold_type=CV_THRESH_BINARY:
dst(x,y) = max_value, if src(x,y)>T(x,y)
           0, otherwise

threshold_type=CV_THRESH_BINARY_INV:
dst(x,y) = 0, if src(x,y)>T(x,y)
           max_value, otherwise
```

where T_I is a threshold calculated individually for each pixel.

For the method `CV_ADAPTIVE_THRESH_MEAN_C` it is a mean of `block_size` × `block_size` pixel neighborhood, subtracted by `param1`.

For the method `CV_ADAPTIVE_THRESH_GAUSSIAN_C` it is a weighted sum (gaussian) of `block_size` × `block_size` pixel neighborhood, subtracted by `param1`.

Why is Face Detection Important?

Face detection is important in our technologically advanced world because it aids in security surveillances and it is also man-power effective and time- effective.

In a surveillance tape or live- stream feed, a face recognition software can be used to find a particular face in a crowd.

A face detecting software can be used to scan the surface of a pool where young children are swimming. If for some reason a child goes under, an alarm will go off because it wouldn't be able to find a face in the image, thereby alerting a rescue party.

The Face detection technique can also be applied to different situations such as motion tracking, whereby a person walking in a parking lot of moving cars can be detected. Or as a security measure whereby a person walking about in a restricted area would be easily detected.

NOTE: THIS IS A REFERENCE EXTRACTED FROM THE WEBSITE BELOW

http://www.geocities.com/jaykapur/face.html

Face Detection in Color Images

Jay P. Kapur
EE499 Capstone Design Project Spring 1997
University of Washington Department of Electrical Engineering

Abstract

This paper presents a technique for automatically detecting human faces in digital color images. The system relies on a two step process which first detects regions which are likely to contain human skin in the color image and then extracts information from these regions which might indicate the location of a face in the image. The skin detection is performed using a skin filter which relies on color and texture information. The face detection is performed on a grayscale image containing only the detected skin areas. A combination of threshholding and mathematical morphology are used to extract object features that would indicate the presence of a face. The face detection process works predictably and fairly reliably, as test results show.

I. Introduction

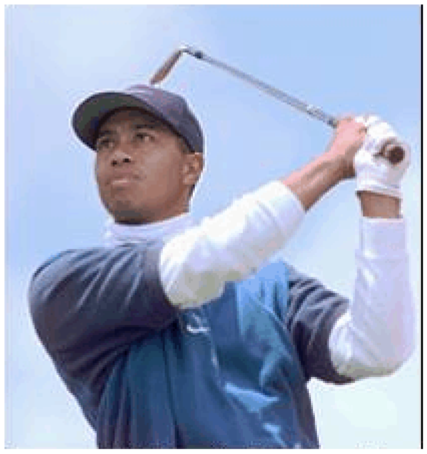
Designing a system for automatic image content recognition is a non-trivial task that has been studied for a variety of applications. Computer recognition of specific objects in digital images has been put to use in manufacturing industries, intelligence and surveillance, and image database cataloging to name a few. In this project, a prototype algorithm for automating the detection of human faces in digital photographs was developed and can serve as an introduction for future work in detecting people in images.

Several systems designed for the purpose of finding people or faces in images have already been proposed by numerous research groups. Some of these programs, such as the Rowley, Baluja, and Kanade system developed at Carnegie Mellon, rely on training of a neural network and computing distance measures between training sets to detect a face. Other software packages exist which can recognize facial features in pictures known to contain a human face somewhere in the image. This project focused on face detection in arbitrary color images and differs from the first type of system in that it relies on a combination of color and grayscale information. Additionally, it does not require the time consuming process of training a neural net or computing distance measures between every possible region in the image. The developed system also differs from those software packages that recognize facial features because, in this scenario, the task is to detect a facial region in an arbitrary image, and not to analyze images known to contain a face.

The process for detection of faces in this project was based on a two-step approach. First, the image is filtered so that only regions likely to contain human skin are marked. This filter was designed using basic mathematical and image processing functions in MATLAB and was based on the skin filter designed for the Berkeley-Iowa Naked People Finder. Modifications to the filter algorithm were made to offer subjective improvement to the output. The second stage involves taking the marked skin regions and removing the darkest and brightest regions from the map. The removed regions have been shown through empirical tests to correspond to those regions in faces which are usually the eyes and eyebrows, nostrils, and mouth. By performing several basic image analysis techniques, the regions with "holes" created by the threshholding can be considered likely to be faces. This second stage was a combination of Khoros visual programming and MATLAB functions. The entire system was entirely automated and required no user intervention save for indicating the correct file names to be processed at each stage. While not implemented in this project, a more advanced program could implement a third step to discriminate between hole sizes and spatial relationships to make an even more robust detection system.

II. Skin Filter

The skin filter is based on the Fleck and Forsyth algorithm with some modifications. For comparison, one might want to consult their report published on the Web at http://www.cs.uiowa.edu/~mfleck/vision-html/naked-skin.html. In this project, the filter was built in MATLAB. Several of the low level image processing functions are already built into the MATLAB environment and this project required time to be invested in building a working filter algorithm, not writing code for low level functions. A description of how the filter operates will be detailed in this section, and the skin filter function as written in MATLAB format is provided in Appendix A.



Original RGB image

The input color image should be in RGB format with color intensity values ranging from 0 to 255. Due to restrictions on speed and performance, this project used images smaller than 250x250 in area. The RGB matrices are "zeroed" to supposedly prevent desaturation when the image is converted from RGB color space to IRgBy color space. The smallest intensity value greater than 10 pixels from any edge in any of the three color planes is set as the zero-response of the image. This value is subtracted from all three color planes.

The RGB image is transformed to log-opponent (IRgBy) values and from these values the texture amplitude, hue, and saturation are computed. The conversion from RGB to log-opponent is calculated according to a variation on the formula given by Fleck& amp;Forsyth:

I= [L(R)+L(B)+L(G)]/3

Rg = L(R)-L(G)

By = L(B)-[L(G)+L(R)]/2

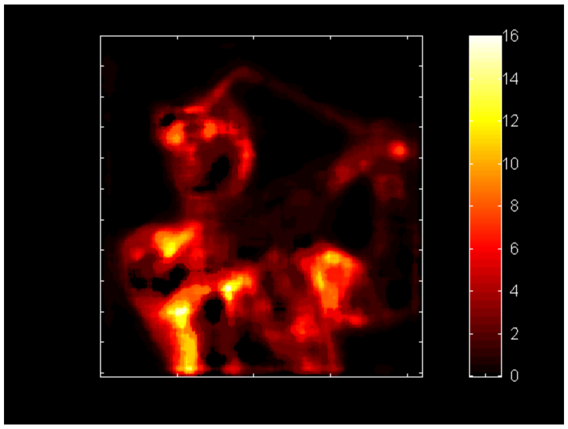
The L(x) operation is defined as L(x)=105*log10(x+1). The Rg and By matrices are then filtered with a windowing median filter of with sides of length 4*SCALE. The SCALE value is calculated as being the closest integer value to (height+width)/320. The m edian filtering is the rate limiting step throughout the skin detection process, and could be improved by implementing an approximation of a windowing median filter as suggested by Fleck's multi-ring operator.

A texture amplitude map is used to find regions of low texture information. Skin in images tends to have very smooth texture and so one of the constraints on detecting skin regions is to select only those regions with little texture. The texture map is generated from the matrix I by the following steps:

1. Median filter I with a window of length 8*SCALE on a side

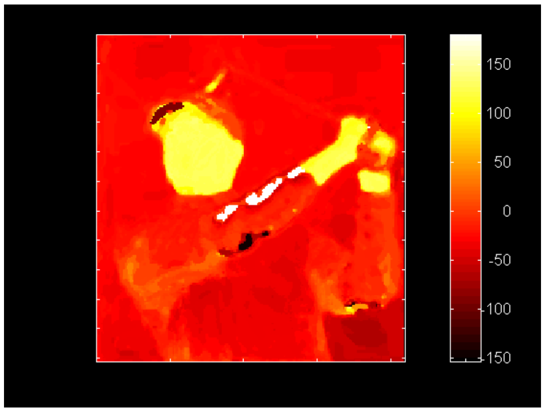
2. Subtract the filtered image from the original I matrix

3. Take the absolute value of the difference and median filter the result with a window of length 12*SCALE on a side.

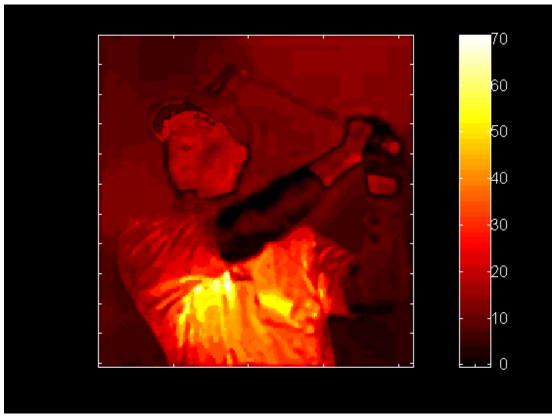


Texture Amplitude Map

Hue and saturation are used to select those regions whose color matches that of skin. The conversion from log opponent to hue is hue = (atan^2(Rg,By)), where the resulting value is in degrees. The conversion from log opponent to saturatio n is saturation = sqrt(Rg^2+By^2). Using constraints on texture amplitude, hue, and saturation, regions of skin can be marked.



Hue Image



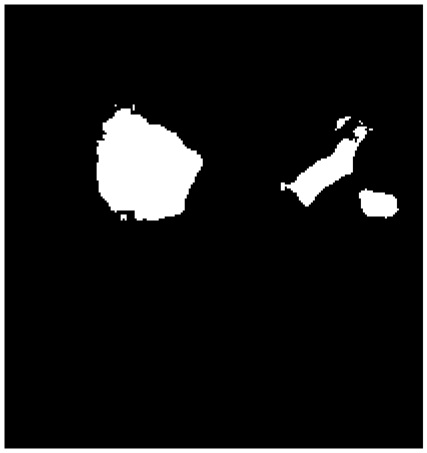
Saturation Image

If a pixel falls into either of two ranges it is marked as being skin in a binary skin map array where 1 corresponds to the coordinates being a skin pixel in the original image and 0 corresponds to a non-skin pixel. The allowed ranges are either :

(1) texture<4.5, 120<HUE<saturation<60<>

(2) texture<4.5, 150<HUE<saturation<80<>

The skin map array can be considered as a black and white binary image with skin regions (value 1) appearing as white. The binary skin map regions are expanded using a dilation operator and a disc structuring element. This helps to enlarge the skin map regions to include skin/background border pixels, regions near hair or other features, or desaturated areas. The dilation adds 8-connected pixels to the edges of objects. In this implementation, the dilation was performed recursively five times for best results. The expanded map regions are then checked against a lenient constraint on hue and saturation values, independent of texture. If a point marked in the skin map corresponds to a pixel with 110<=hue<=180 and 0<=saturation<=130, the value remains 1 in the map.



Skin Map

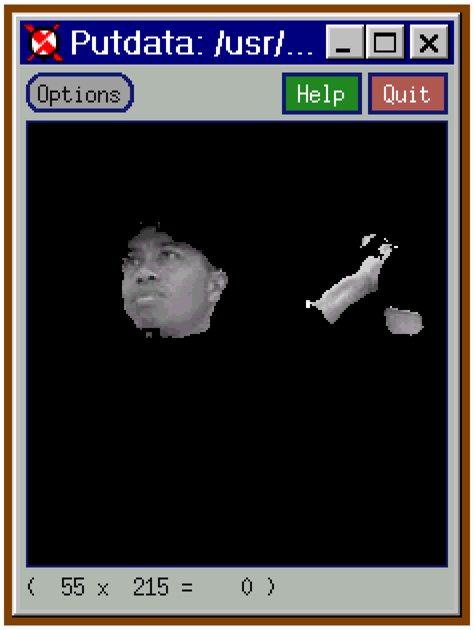
The skin filter is not perfect, either due to coding errors or improper constraints, because there is a tendency for highly saturated reds and yellows to be detected as skin. Often this causes problems in the face detection when a large red or yellow p atterned object is present in the image. See the results in Appendix B for several examples of the skin filter output and cases in which the skin filter marked highly saturated red and yellow as skin.

III. Face Detection From Skin Regions

The binary skin map and the original image together are used to detect faces in the image. The technique relies on threshholding the skin regions properly so that holes in face regions will appear at the eyebrows, eyes, mouth, or nose. Theoretically, all other regions of skin will have little or no features and no holes will be created except for at the desired facial features. This method seems to be an oversimplification of the problem, but with some additional constraints on hole sizes or spatial relationships, could prove to be a powerful, fast, and simple alternative to neural network processes.

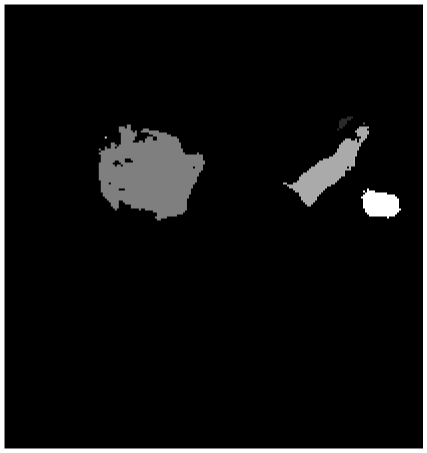
Detection of face regions was broken into two parts, the first using the Khoros visual programming application and the second part using a MATLAB program. The two Khoros workspaces are shown in Appendix A along with the MATLAB code. All of the functions used are standard image analysis techniques (hole filling algorithms, threshholding, connected components labeling, etc.) that should be straightforward, though perhaps tedious, to build in any programming language.

The first step is to ensure that the binary skin map is made up of solid regions (i.e. no holes). Closing holes in the skin map is important because later the program assumes that the only holes are those generated after the threshholding operation. A hole closing is performed on the skin map image with a 3x3 disc structuring element and then this image is multiplied by a grayscale conversion of the original image. The result is a grayscale intensity image showing only the parts of the image containing skin.



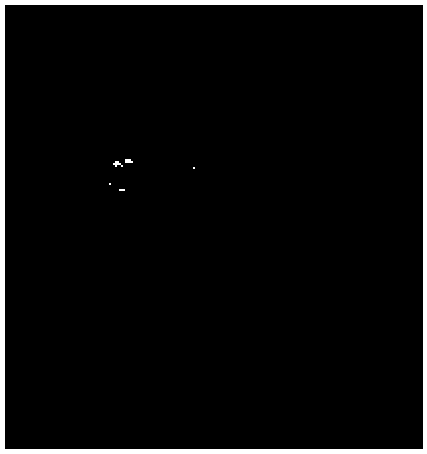
Skin Map Multiplied by Grayscale Image

To improve contrast, a histogram stretch is performed on the resulting grayscale image. This helps to make the dark and light regions fall into more predictable intensity ranges and compensates somewhat for effects of illumination in the image. The image can now be threshholded to remove the darkest and lightest pixels. Experimentation showed that an acceptable threshold was to set all pixels with values between 95 and 240 equal to 1 and those pixels above and below the cutoff equal to 0. For most test images, this threshold work quite well. The binary image created by the threshold is then passed through a connected components labeling to generate a "positive" image showing distinct skin regions.



Positive Labeled Image

A negative image is next generated that will show only holes as objects. Hole closing of the binary image generated by the threshold operation is performed with a 4x4 disc structuring element. The result is subtracted from the original binary image and the difference shows only hole objects.



Negative 'Hole' Image

The negative hole image and the positive labeled image are then used together to find which objects in the image might be faces. First those holes in the negative image which are only 1 pixel in size are removed because these tend to represent anomalous holes. An even better technique might be to remove all but the three largest hole objects from the negative image. The hole objects are expanded using a dilation and this binary image is then multiplied by the positive labeled image. The product is an image where only the pixels surrounding a hole are present. Because the positive image was labeled, the program can easily determine which objects have holes, and which do not. A simple function computes which integers appear in the hole adjacency image and then generates an output image containing the labeled connected components that have this value.

Face Objects

Because this process relies only on finding holes in threshholded objects, there is a greater chance of finding faces regardless of the perspective. A drawback is that there is also a greater risk of detecting non-face objects. The test results show very good performance when a face occupies a large portion of the image, and reasonable performance on those images depicting people as part of a larger scene. To make the program more robust, detected face objects could be rejected if they don't occupy a significant area in the image. Another drawback of this process is that images in which people appear partially clothed tend will result in a very large skin map. The result is often a labeling of the entire head, arms, and torso as a single object. Thus the face finding is an overestimate of potential skin objects. All things considered, this technique developed over a period of ten weeks shows promise and with some "intelligence" added to the algorithm, could generate very reliable results on a wide variety of images.

Appendix A

Matlab Code
This code is used for the skin filter and the final face region detection. The Khoros workspaces are long gone, but it is easy enough to write Matlab code to do the same thing.

skinfilt.m
skinmap.m
logopp.m
face.m
fscript.m

A picture is worth more than a thousand words.

We have heard this expression countless times. A picture gives a much clearer impression of a situation or an object than numerous descriptions. Having an accurate visual perceptive of things has a high social, economic and technical value that is why image analysis (a.k.a. image understanding), image processing and computer vision plays a huge role in the research industry today.

Digital image processing stems from two different application areas:

1) Improvement of pictorial information for human interpretation
2) Processing of image data for storage, transmission and representation of autonomous machine perception.

<u>What is Digital Image Processing?</u>

The field of digital image processing refers to processing digital images by means of a digital computer. A digital image composes of a finite number of elements which have a particular location and value and these elements are referred to as ***picture*** elements, ***image*** elements, ***pels*** and ***pixels***.

Images are referred to more than just the projections generated by the visual band of the EM (electromagnetic) waves apparent to humans. Images generated from the entire band of the EM waves ranging from gamma to radio waves can be perceived by imaging machines. Some of these images include ultrasound, electron microscopy, and computer-generated images.

There are **3** computerized processing levels:

1) Low-level process: - is characterized by the fact that both the inputs and outputs are images. These involve primitive operations such as image preprocessing to reduce noise, contrast enhancement and image sharpening.
2) Mid-level process: - is characterized by the fact that its inputs generally are images, but its outputs are attributes extracted from those images such as, edges, contours, and the identity of individual objects. Mid-level processing on images involves tasks such as segmentation (partitioning an image into regions or objects), description of those objects to reduce them to a form suitable for computer processing, and classification (recognition) of individual objects.
3) High-level process: - involves trying deduce an ensemble of recognized objects, from image analysis to performing the cognitive function usually associated with vision.

Origins of Digital Image Processing

In the early 1920's the Bartlane cable picture transmission system was introduced, thereby reducing the transportation time for a picture from New York to England by days. Digital images were first applied in the newspaper industry when pictures were first set by submarine cable between London and New York, then the Bartlane cable was introduced reducing transmission time from three weeks to three hours.

There were different phases in technology improvement; in 1921 the method used for receiving images through a coded tape by a telegraph printer was abandoned in favor of a technique based on photographic reproduction made from tapes perforated at the telegraph receiving terminal with evident improvement in both tonal quality and resolution.

The history of digital image processing is intimately tied to the development of the digital computer. The first computers powerful enough to carry out meaningful image processing tasks appeared in the early 1960s. This was when there was significant development of the high-level programming languages COBOL (common business-oriented language) and FORTRAN (formula translator) and the development of operating systems. The birth of digital image processing can be traced to the availability of advanced computers and the onset of the space program during that period. Work on using computer techniques for improving images from a space probe began at the Jet Propulsion Laboratory in Pasadena, California in 1964 when pictures of the moon transmitted by RANGER 7 were processed by a computer to correct various types of image distortion inherent in the on-board television camera.

Fields that Use Digital Image Processing

In parallel with space applications, digital image processing techniques in the late 1960s and early 1970s to be used in (1) medical imaging, (2) remote Earth resources observations and (3) astronomy.

(1) Medical imaging-- The invention in the early 1970s of computerized axial tomography (CAT) is one of the most important events in the application of image processing in medical diagnosis. Tomography consists of algorithms that use the sensed data to construct an image that represents a slice through the object which compose a three-dimensional (3-D) version of the inside of the object.

Tomography was invented independently by Sir Godfrey N. Hounsfield and Professor Allan M. Cormack, who shared the 1979 Nobel Prize in Medicine for their invention. X-rays were discovered in 1895 by Wilhelm Conrad Roentgen, for which he received the 1901 Nobel Prize in Physics. These two inventions, nearly 100 years apart led to some of the most active application areas of image processing today.

Computer procedures are also used to enhance the contrast or code the intensity levels into color for easier interpretation of X-rays and other images used in industry, medicine, and the biological sciences.

(2) Remote earth resources and observations-- Geographers use the same or similar techniques to study pollution patterns from aerial and satellite imagery. Image enhancement and restoration procedures are used to process degraded images of unrecoverable objects or experimental results too expensive to duplicate. In archaeology, image processing methods have successfully restored blurred pictures that were the only available records of rare artifacts lost or damaged after being photographed.

In physics and other related fields, computer techniques routinely enhance images of experiments in areas such as high-energy plasmas and electron microscopy. Similarly successful applications of image processing concepts can be found in astronomy, biology, nuclear medicine, law enforcement defense, and industrial applications.

These examples illustrate processing results intended for human interpretation. The second major area of application of digital image processing techniques deals with machine perception. In this case interest focuses on procedures for extracting from an image, information in a form suitable for computer processing. Examples of the type of information used in machine perception are statistical moments, Fourier transform coefficients, and multidimensional distance measures. Typical problems in machine perception that routinely utilize image processing techniques are automatic character recognition, industrial machine vision for product assembly and inspection, military recognizance, automatic processing of fingerprints, screening of X-rays and blood samples, and machine processing of aerial and satellite imagery for weather prediction and environmental assessment.

Fundamental Steps in Digital Image Processing

Image acquisition- is the first process which involves preprocessing such as scaling.

Image enhancement- this is bringing out obscured detail or highlighting certain features of interest in an image. This technique deals with a number of mathematical functions such as the Fourier Transform.

Image restoration- it improves the appearance of an image but is objective in the sense that this technique tends to be based on mathematical or probabilistic models of image degradation.

Color image processing- this is used as a basis for extracting features of interest in an image.

Wavelets- are the foundation for representing images in various degrees of resolution.

Compression- deals with techniques for reducing the storage required to save an image, or the bandwidth required to transmit it.

Morphological processing- deals with tools for extracting image components that are useful in the representation and description of shape.

Segmentation- partitions an image into its constituent parts or objects.

Representation and description- representation is necessary for transforming raw data into a form suitable for subsequent computer processing. Description, also known as feature selection, deals with extracting attributes that result in some quantitative information of interest.

Recognition- assigns a label to an object based on its descriptors.

Feature Extraction- this is an area of image processing which involves using algorithms to detect and isolate various desired portions of a digitized image or video stream.

Feature Extraction/Detection

This is an area of image processing that uses algorithms to detect and isolate varius desired portions of a digitized image.

Feature Extraction Techniques

Hough Transform-

This identifies lines in an image as well as arbitrary shapes. The purpose of the transform is to determine which of these theoretical lines pass through most features in an image. The input to a Hough transform is usually one to which some kind of edge detection has been applied instead of a raw image.

Each line is represented by two parameters, commonly called ***r*** and ***θ*** which represent the length and angle from the origin of a normal to the line in question. In other words, a line is said to be 90° from ***θ*** and ***r*** units away from the origin at its closest point. By calculating the value of r for every possible value of ***θ***, a sinusoidal curve is created which is unique to that point. This representation of the two parameters is sometimes referred to as ***Hough space***. Thus the points to be transformed are likely to lie on an '**edge**' in the image. The transform itself is quantized into an arbitrary number of ***bins***, each representing an approximate definition of a possible line. Each significant point (or feature) in the edge detected is said to ***vote*** for a set of bins corresponding to the lines that pass through it. By simply incrementing the value stored in each bin for every feature lying on that line, an array is built up which shows which lines fit most closely to the data in the image.

Hough transform of curves, and Generalized Hough transform

The transform described above applies to finding straight lines; a circle for instance can be transformed into a set of three parameters representing its center and radius, so that the Hough space becomes three dimensional. Arbitrary ellipses, curves and shapes expressed as a set of parameters can be found this way. For more complicated shapes, the Generalized Hough transform is used, which allows a feature to vote for particular position, orientation and/scaling or the shape using a predefined look-up table.

Using Weighted Features

The Hough transform accounts for uncertainty in the underlying detection of edges by allowing features to vote with varying weight.

Hierarchical Hough Transform

A final enhancement that is sometimes effective is to perform a hierarchical set of Hough transform on the same image, using progressively smaller bins. If the image is first analyzed using a small number of bins, each representing a large range of potential lines, the most likely of these can then be analyzed in more detail. That is finding the bins with the highest count in one stage can be used to constrain the range of values searched in the text.

Image Feature Detection based on Scale-Interaction Model

This feature detector is responsive to short lines, line endings, corners and other sharp changes in curvature. Features are locations in an image that are perceptually interesting. Previous work on feature detection includes the use of grey level statistics and the detection of edges and corners.

Detecting edges and corners are particularly useful in analysis of aerial images of urban scenes, airport facilities, image to map matching, etc.

Algorithms based on grey level statistics are applicable to a wider variety of images such as desert scenes and vegetation and images which don’t necessarily contain man made structures.

DICTIONARY

Raster Image :-
An image that is composed of pixel patterns, also known as **Bitmapped Image**

Pixel :-
It is the smallest complete element of an Image.
It is a picture element. The quality of an image depends on the number of pixels per inch that make up that image.