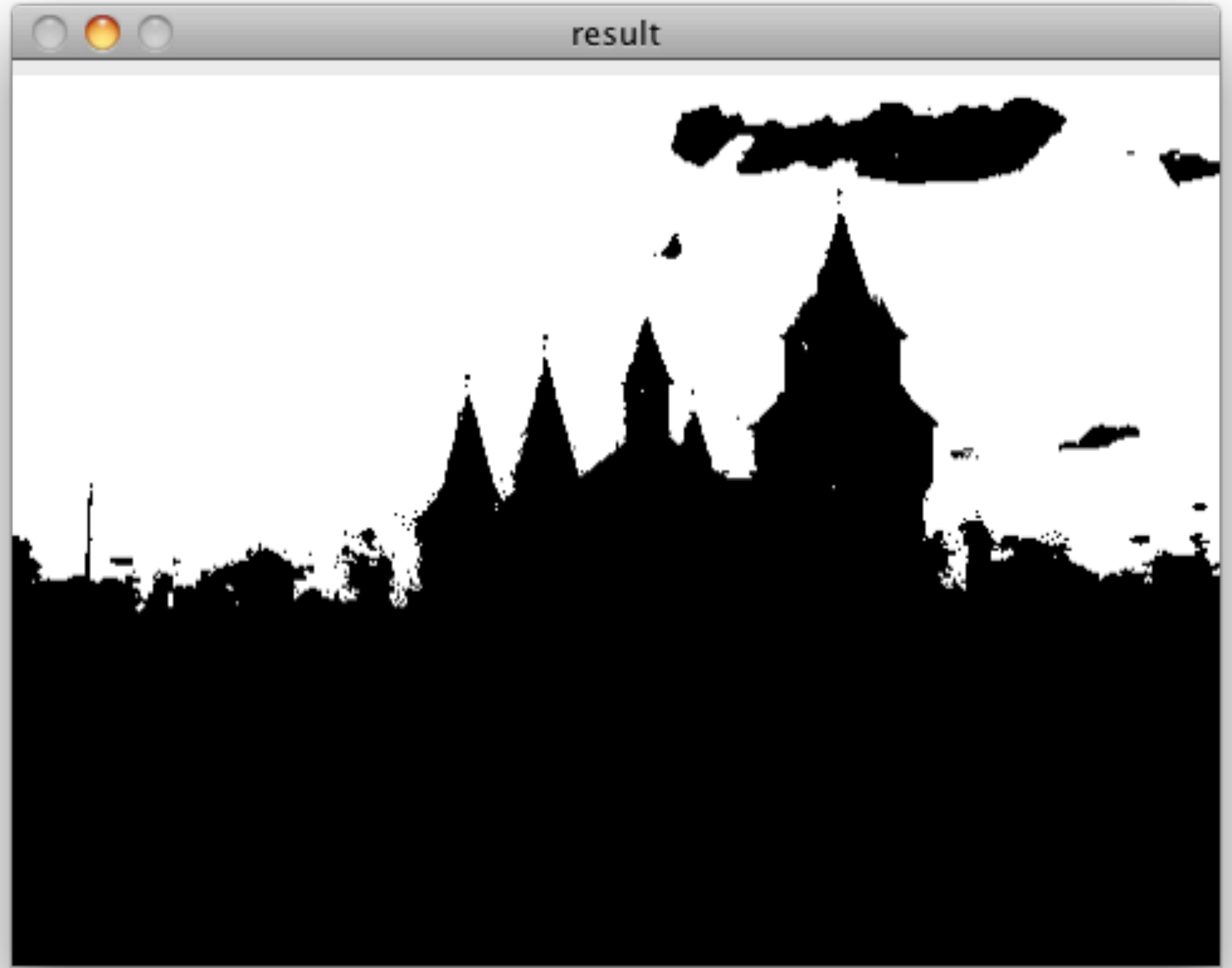
3

Processing Images with Classes

Color Detection

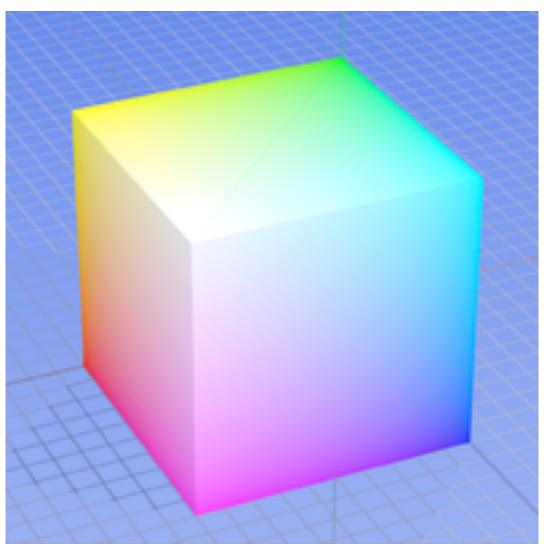
Given a color value, the black area represents those pixels of similar color.



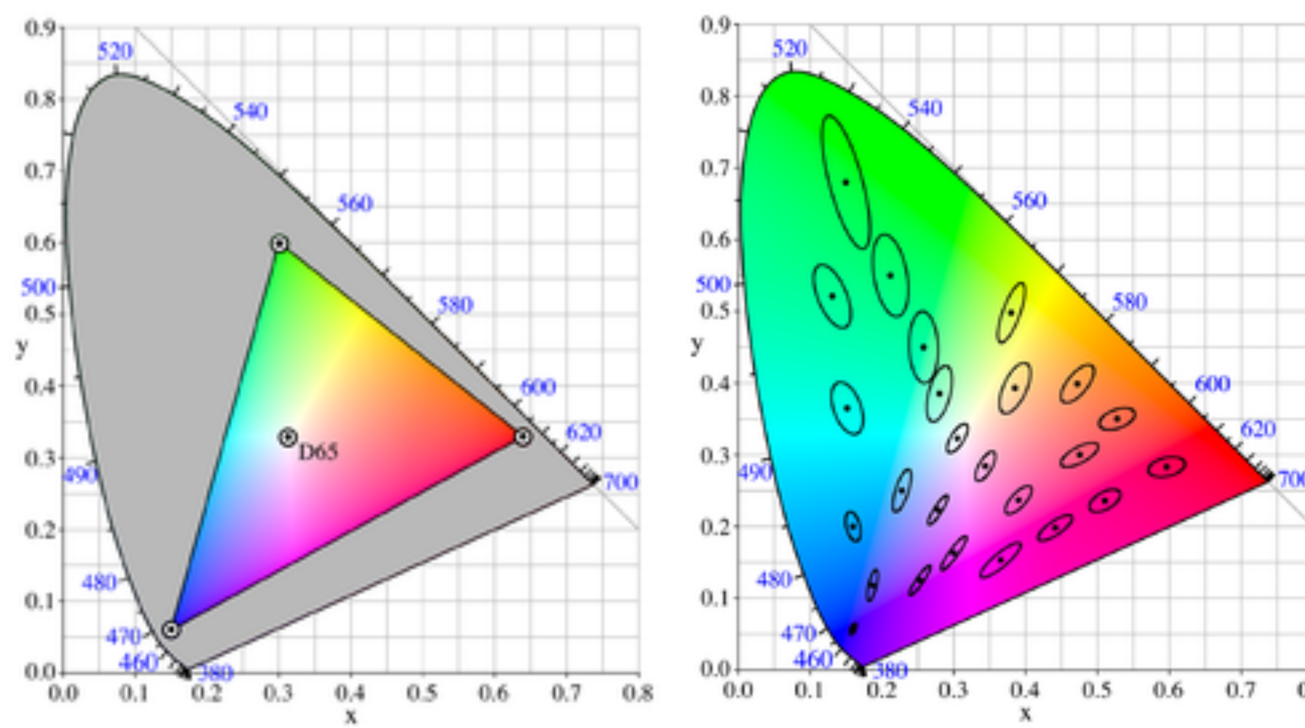
keywords: target color, threshold, color-space distance

Color Space Distance, Distance in Multidimensional Space

$$
\begin{array}{cc}
L_2 \text{ distance} & \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2} \\
L_1 \text{ distance} & |x_1 - y_1| + |x_2 - y_2| + |x_3 - y_3| \\
L_\infty \text{ distance} & \max\{|x_1 - y_1|, |x_2 - y_2|, |x_3 - y_3|\}
\end{array}
$$



The RGB color model mapped to a cube. The horizontal x-axis as red values increasing to the left, y-axis as blue increasing to the lower right and the vertical z-axis as green increasing towards the top. The origin, black, is hidden behind the cube.



http://en.wikipedia.org/wiki/Color_difference

colorDetection.cpp

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>

#include "colordetector.h"

int main()
{
    // Create image processor object
    ColorDetector cdetect;

    // Read input image
    cv::Mat image= cv::imread("boldt.jpg");
    if (!image.data)
      return 0;

    // set input parameters
    cdetect.setTargetColor(130,190,230); // here blue sky

    // Read image, process it and display the result
    cv::Mat result = cdetect.process(image);

    cv::namedWindow("result");
    cv::imshow("result", result);
    cv::waitKey();

    return 0;
}
```

colordetector.h

```
class ColorDetector {
private:
    // minimum acceptable distance
    int minDist;
    // target color
    cv::Vec3b target;
    // image containing resulting binary map
    cv::Mat result;
    // inline private member function
    // Computes the distance from target color.
    int getDistance(const cv::Vec3b& color) const
    {
        return abs(color[0]-target[0])+
               abs(color[1]-target[1])+
               abs(color[2]-target[2]);
    }

public:
    // empty constructor
    ColorDetector() : minDist(100) {
        // default parameter initialization here
        target[0]= target[1]= target[2]= 0;
    }
```

colordetector.h

```
        // Getters and setters

        // Sets the color distance threshold.
        // Threshold must be positive, otherwise distance threshold
        // is set to 0.
        void setColorDistanceThreshold(int distance) {
            if (distance<0)
                distance=0;
            minDist= distance;
        }
        // Gets the color distance threshold
        int getColorDistanceThreshold() const {
            return minDist;
        }
        // Sets the color to be detected
        void setTargetColor(unsigned char red, unsigned char green, unsigned char blue) {
            target[2]= red;
            target[1]= green;
            target[0]= blue;
        }
        // Sets the color to be detected
        void setTargetColor(cv::Vec3b color) {
            target= color;
        }
        // Gets the color to be detected
        cv::Vec3b getTargetColor() const {
            return target;
        }
        // Processes the image. Returns a 1-channel binary image.
        cv::Mat process(const cv::Mat &image);
    }; // class ColorDetector
```

colordetector.cpp

```
#include "colordetector.h"

cv::Mat ColorDetector::process(cv::Mat &image)
{
    result.create(image.rows,image.cols,CV_8U);

    int ncols = image.cols * image.channels();

    for (int r=0; r<image.rows; r++)
        {
        uchar *p_rgb = image.ptr<uchar>(r);
        uchar *result_bptr = result.ptr<uchar>(r);

        for (int c=0; c < image.cols; c++, p_rgb+=3)
            {
            cv::Vec3b rgb(p_rgb[0], p_rgb[1], p_rgb[2]);

            if (getDistance(rgb)<minDist)
                result_bptr[c] = 255;
            else
                result_bptr[c] = 0;
            }
        }

     return result;
}
```

colordetector.cpp

```cpp
#include "colordetector.h"

cv::Mat ColorDetector::process(const cv::Mat &image)
{
      // re-allocate binary map if necessary
      // same size as input image, but 1-channel
      result.create(image.rows, image.cols, CV_8U);

      // get the iterators
      cv::Mat_<cv::Vec3b>::const_iterator it= image.begin<cv::Vec3b>();
      cv::Mat_<cv::Vec3b>::const_iterator itend= image.end<cv::Vec3b>();
      cv::Mat_<uchar>::iterator itout= result.begin<uchar>();

      // for each pixel
      for ( ; it!= itend; ++it, ++itout) {

      // process each pixel ---------------------

        // compute distance from target color
        if (getDistance(*it)<minDist)
           *itout= 255;
        else
           *itout= 0;

        // end of pixel processing ----------------
      }
      return result;
}
```

4

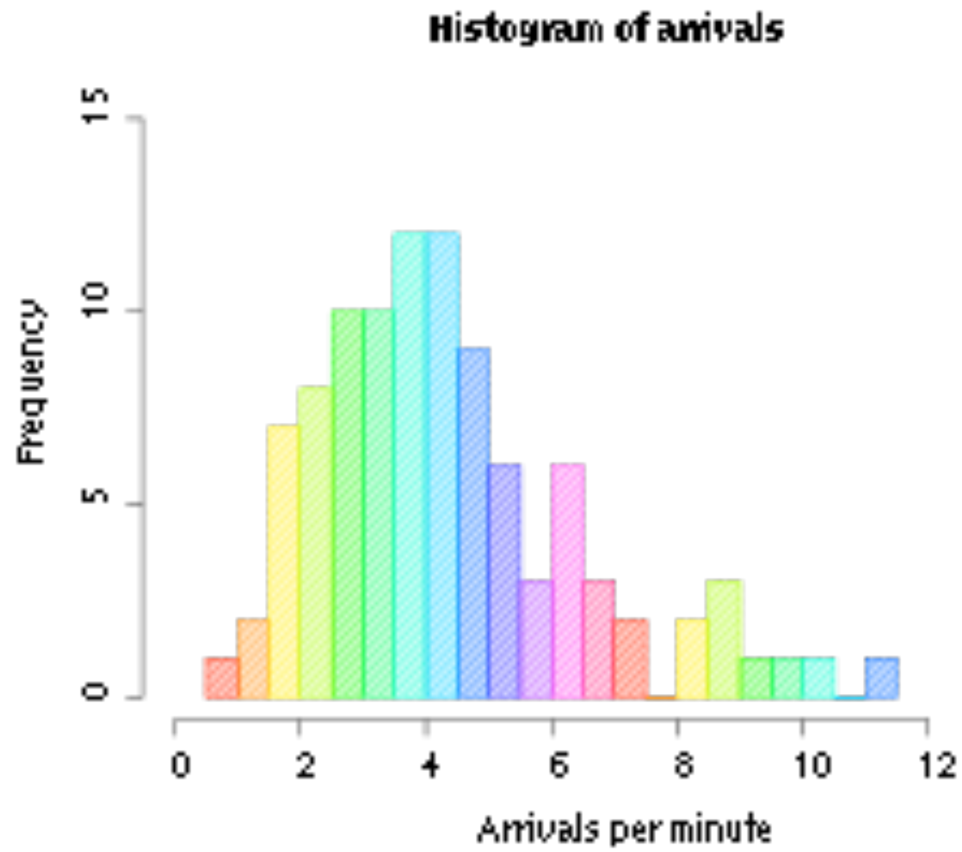
Counting the Pixels with Histograms

In this chapter, we will cover:

- Computing the image histogram
- Applying look-up tables to modify image appearance
- Equalizing the image histogram
- Backprojecting a histogram to detect specific image content
- Using the mean shift algorithm to find an object
- Retrieving similar images using histogram comparison

What is Histogram?

In statistics, a **histogram** is a graphical representation showing a visual impression of the distribution of data. It is an estimate of the probability distribution of a continuous variable and was first introduced by Karl Pearson.[1]



Let's make a histogram!

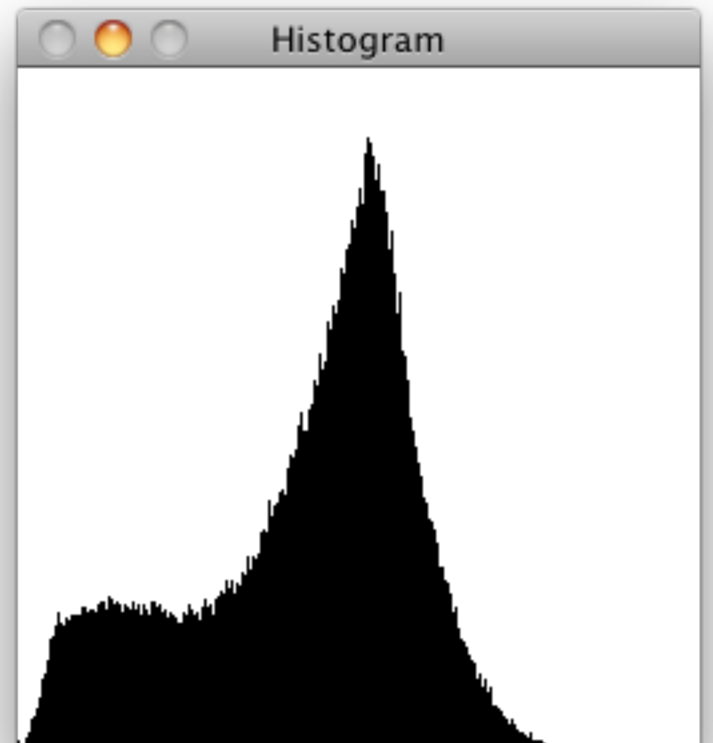
```cpp
// file: histograms.cpp
#include <iostream>
using namespace std;

#include "opencv2/highgui/highgui.hpp"
#include "histogram.h"

int main()
{
    // Read input image
    cv::Mat image= cv::imread("group.jpg",0);
    if (!image.data)
       return 0;

    // Display the image
    cv::namedWindow("Image");
    cv::imshow("Image",image);

    // The histogram object
    Histogram1D h;

    // Compute the histogram
    cv::Mat histo= h.getHistogram(image);

    // Loop over each bin
    for (int i=0; i<256; i++)
       cout << "Value " << i << " = " << histo.at<float>(i) << endl;

    // Display a histogram as an image
    cv::namedWindow("Histogram");
    cv::Mat himage = h.getHistogramImage(image);
    cv::imshow("Histogram", himage);
    std::cerr << "Histogram Displayed" << endl;
```

histogram.h

```
class Histogram1D {
private:
    int histSize[1];
    float hranges[2];
    const float* ranges[1];
    int channels[1];

public:
    Histogram1D() {
        // Prepare arguments for 1D histogram
        histSize[0]= 256;
        hranges[0]= 0.0;
        hranges[1]= 255.0;
        ranges[0]= hranges;
        channels[0]= 0; // by default, we look at channel 0
    }

    // Computes the 1D histogram.
    cv::Mat getHistogram(const cv::Mat &image) {
       cv::Mat hist;
       // Compute histogram
       cv::calcHist(&image,
                    1,       // histogram of 1 image only
                    channels,   // the channel used
                    cv::Mat(),  // no mask is used
                    hist,    // the resulting histogram
                    1,       // it is a 1D histogram
                    histSize,   // number of bins
                    ranges      // pixel value range
                    );

       return hist;
    }
...
```

```
// Computes the 1D histogram and returns an image of it.
cv::Mat getHistogramImage(const cv::Mat &image) {
   // Compute histogram first
   cv::Mat hist= getHistogram(image);

   // Get min and max bin values
   double maxVal=0;
   double minVal=0;
   cv::minMaxLoc(hist, &minVal, &maxVal, 0, 0);

   // Image on which to display histogram
   cv::Mat histImg(histSize[0], histSize[0], CV_8U,cv::Scalar(255));

   // set highest point at 90% of nbins
   int hpt = static_cast<int>(0.9*histSize[0]);

   // Draw vertical line for each bin
   for( int h = 0; h < histSize[0]; h++ ) {

      float binVal = hist.at<float>(h);
      int intensity = static_cast<int>(binVal*hpt/maxVal);
      cv::line(histImg,
               cv::Point(h,histSize[0]),
               cv::Point(h,histSize[0]-intensity),
                cv::Scalar::all(0)
              );
   }

   return histImg;
}
```


Image


Histogram

cv::line

Draws a line segment connecting two points.

C++: void **line**(Mat& **img**,
Point **pt1**, Point **pt2**,
const Scalar& **color**,
int **thickness**=1, int**lineType**=8, int **shift**=0)

Parameters:

1 **img** – Image.
2 **pt1** – First point of the line segment.
3 **pt2** – Second point of the line segment.
4 **color** – Line color.
5 **thickness** – Line thickness.
6 **lineType** –Type of the line:
  - **8** (or omitted) - 8-connected line.
  - **4** - 4-connected line.
  - **CV_AA** - antialiased line.

7 **shift** – Number of fractional bits in the point coordinates.

This is equivalent to a 4-vector datatype.

Scalar_

class **Scalar_**

Template class for a 4-element vector derived from Vec.

```
template<typename _Tp> class Scalar_ : public Vec<_Tp, 4> { ... };

typedef Scalar_<double> Scalar;
```

```
// make a 7x7 complex matrix filled with 1+3j.
Mat M(7,7,CV_32FC2,Scalar(1,3));


// create a 100x100x100 8-bit multi-dimensional array
int sz[] = {100, 100, 100};
Mat bigCube(3, sz, CV_8U, Scalar::all(0));


// create a new 320x240 BGR image
Mat img(Size(320,240),CV_8UC3);
// select a ROI
Mat roi(img, Rect(10,10,100,100));
// fill the ROI with (0,255,0) (which is green in RGB space);
// the original 320x240 image will be modified
roi = Scalar(0,255,0);
```

/usr/local/include/opencv2/core/core.hpp

```
namespace cv {

#undef abs
#undef min
#undef max
#undef Complex

    using std::vector;
    using std::string;
    using std::ptrdiff_t;

    template<typename _Tp> class CV_EXPORTS Size_;
    template<typename _Tp> class CV_EXPORTS Point_;
    template<typename _Tp> class CV_EXPORTS Rect_;
    template<typename _Tp, int cn> class CV_EXPORTS Vec;
    template<typename _Tp, int m, int n> class CV_EXPORTS Matx;

    typedef std::string String;
    typedef std::basic_string<wchar_t> WString;

    class Mat;
    typedef Mat MatND;   // MatND is Mat!
    class SparseMat;
```

Mat::Mat

Various Mat constructors

C++: `Mat::Mat()`

C++: `Mat::Mat(int rows, int cols, int type)`

C++: `Mat::Mat(Size size, int type)`

C++: `Mat::Mat(int rows, int cols, int type, const Scalar& s)`

C++: `Mat::Mat(int ndims, const int* sizes, int type)`

C++: `Mat::Mat(int ndims, const int* sizes, int type, const Scalar& s)`

C++: `Mat::Mat(int ndims, const int* sizes, int type, void* data, const size_t* steps=0)`

Parameters:

1. **ndims** – Array dimensionality.
2. **rows** – Number of rows in a 2D array.
3. **cols** – Number of columns in a 2D array.
4. **size** – 2D array size: `Size(cols, rows)` . In the `Size()` constructor, the number of rows and the number of columns go in the reverse order.
5. **sizes** – Array of integers specifying an n-dimensional array shape.
6. **type** – Array type. Use `CV_8UC1, ..., CV_64FC4` to create 1-4 channel matrices, or `CV_8UC(n),..., CV_64FC(n)` to create multi-channel (up to `CV_MAX_CN` channels) matrices.

http://opencv.itseez.com/modules/core/doc/basic_structures.html#matx

calcHist

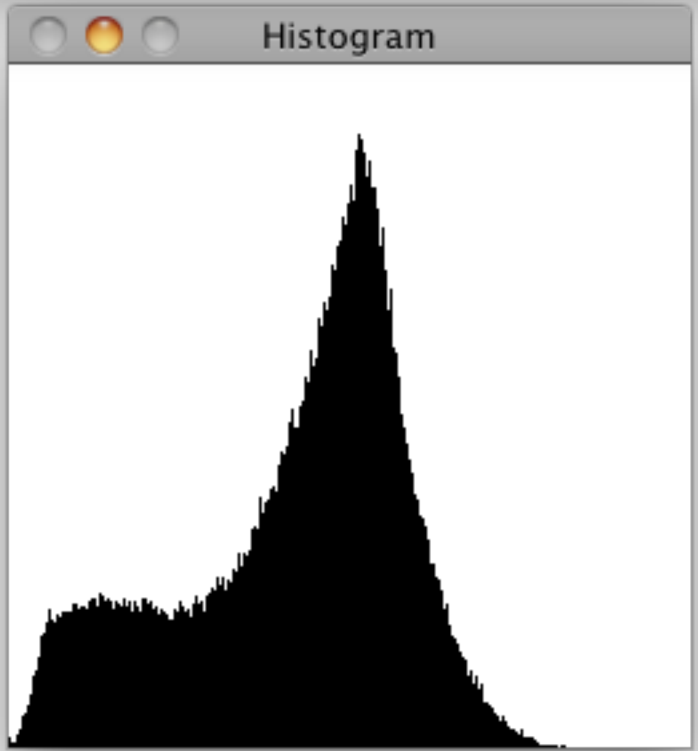
Calculates a histogram of a set of arrays.

C++: void **calcHist**(const Mat* **arrays**, int **narrays**, const int* **channels**, InputArray **mask**, OutputArray **hist**, int **dims**, const int* **histSize**, const float** **ranges**, bool **uniform**=true, bool **accumulate**=false)

Parameters:

1. **arrays** – Source arrays. They all should have the same depth, `CV_8U` or`CV_32F` , and the same size. Each of them can have an arbitrary number of channels.
2. **narrays** – Number of source arrays.
3. **channels** – List of the `dims` channels used to compute the histogram. The first array channels are numerated from 0 to `arrays[0].channels()-1` , the second array channels are counted from `arrays[0].channels()` to`arrays[0].channels() + arrays[1].channels()-1`, and so on.
4. **mask** – Optional mask. If the matrix is not empty, it must be an 8-bit array of the same size as `arrays[i]` . The non-zero mask elements mark the array elements counted in the histogram.
5. **hist** – Output histogram, which is a dense or sparse `dims` -dimensional array.
6. **dims** – Histogram dimensionality that must be positive and not greater than`CV_MAX_DIMS` (equal to 32 in the current OpenCV version).
7. **histSize** – Array of histogram sizes in each dimension.
8. **ranges** – Array of the `dims` arrays of the histogram bin boundaries in each dimension.

```
   // creating a binary image by thresholding at the valley
cv::Mat thresholded;
cv::threshold(image,thresholded,60,255,cv::THRESH_BINARY);
```



threshold

Applies a fixed-level threshold to each array element.

C++: double **threshold**(InputArray **src**, OutputArray **dst**, double **thresh**, double **maxVal**, int **thresholdType**)

Parameters:

1. **src** – Source array (single-channel, 8-bit of 32-bit floating point).
2. **dst** – Destination array of the same size and type as `src` .
3. **thresh** – Threshold value.
4. **maxVal** – Maximum value to use with the `THRESH_BINARY` and`THRESH_BINARY_INV` thresholding types.
5. **thresholdType** – Thresholding type (see the details below).

The function applies fixed-level thresholding to a single-channel array. The function is typically used to get a bi-level (binary) image out of a grayscale image or for removing a noise, that is, filtering out pixels with too small or too large values. There are several types of thresholding supported by the function. They are determined by `thresholdType` :

- **THRESH_BINARY**

$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxVal} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

- **THRESH_BINARY_INV**

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{maxVal} & \text{otherwise} \end{cases}$$

- **THRESH_TRUNC**

$$\texttt{dst}(x,y) = \begin{cases} \texttt{threshold} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

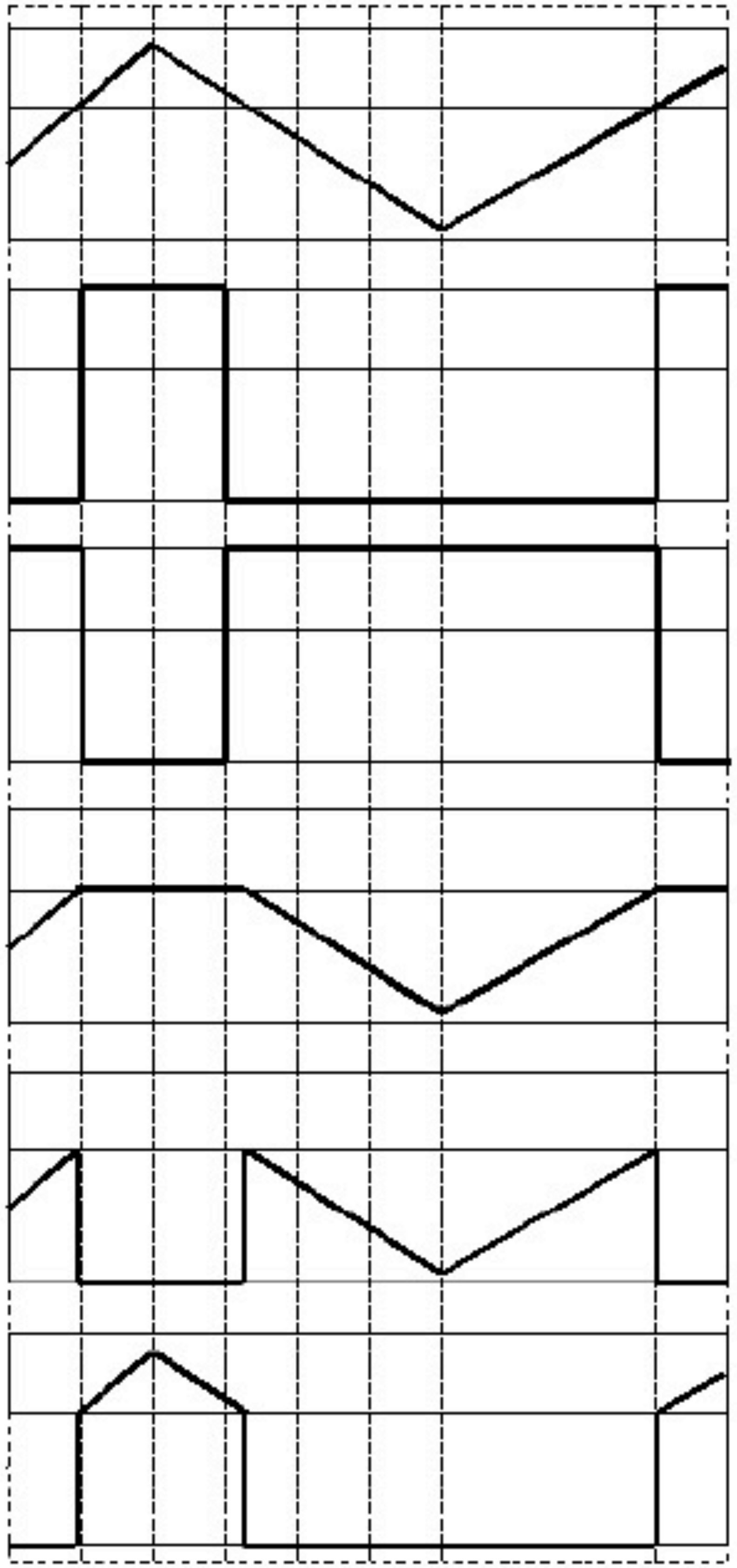
- **THRESH_TOZERO**

$$\texttt{dst}(x,y) = \begin{cases} \texttt{src}(x,y) & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

- **THRESH_TOZERO_INV**

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

Al the special value `THRESH_OTSU` may be combined with one of the above values. In this case, the function determines the optimal threshold value using the Otsu's algorithm and uses it instead of the specified `thresh` . The function returns the computed threshold value. Currently, the Otsu's method is implemented only for 8-bit images.



Value and Threshold Level	
Threshold Binary	$$\mathtt{dst}(x,y) = \begin{cases} \mathtt{maxVal} & \text{if } \mathtt{src}(x,y) > \mathtt{thresh} \\ 0 & \text{otherwise} \end{cases}$$
Threshold Binary, Inverted	$$\mathtt{dst}(x,y) = \begin{cases} 0 & \text{if } \mathtt{src}(x,y) > \mathtt{thresh} \\ \mathtt{maxVal} & \text{otherwise} \end{cases}$$
Truncate	$$\mathtt{dst}(x,y) = \begin{cases} \mathtt{threshold} & \text{if } \mathtt{src}(x,y) > \mathtt{thresh} \\ \mathtt{src}(x,y) & \text{otherwise} \end{cases}$$
Threshold to Zero, Inverted	$$\mathtt{dst}(x,y) = \begin{cases} \mathtt{src}(x,y) & \text{if } \mathtt{src}(x,y) > \mathtt{thresh} \\ 0 & \text{otherwise} \end{cases}$$
Threshold to Zero	$$\mathtt{dst}(x,y) = \begin{cases} 0 & \text{if } \mathtt{src}(x,y) > \mathtt{thresh} \\ \mathtt{src}(x,y) & \text{otherwise} \end{cases}$$

Otsu's threshold selection algorithm

Method

In Otsu's method we exhaustively search for the threshold that minimizes the intra-class variance, defined as a weighted sum of variances of the two classes:

$$\sigma_w^2(t) = \omega_1(t)\sigma_1^2(t) + \omega_2(t)\sigma_2^2(t)$$

Weights ω_i are the probabilities of the two classes separated by a threshold t and σ_i^2 variances of these classes.

Otsu shows that minimizing the intra-class variance is the same as maximizing inter-class variance:[2]

$$\sigma_b^2(t) = \sigma^2 - \sigma_w^2(t) = \omega_1(t)\omega_2(t)\left[\mu_1(t) - \mu_2(t)\right]^2$$

which is expressed in terms of class probabilities ω_i and class means μ_i which in turn can be updated iteratively. This idea yields an effective algorithm.

[edit]

Algorithm

1. Compute histogram and probabilities of each intensity level
2. Set up initial $\omega_i(0)$ and $\mu_i(0)$
3. Step through all possible thresholds maximum intensity
   1. Update ω_i and μ_i
   2. Compute $\sigma_b^2(t)$
4. Desired threshold corresponds to the maximum $\sigma_b^2(t)$

[edit]

http://en.wikipedia.org/wiki/Otsu's_method