In computer vision and image processing, **Otsu's method** is used to automatically perform histogram shape-based image thresholding, or, the reduction of a graylevel image to a binary image.

The algorithm assumes that the image to be thresholded contains two classes of pixels (e.g. foreground and background) then calculates the optimum threshold separating those two classes so that their combined spread (intra-class variance) is minimal.

Let the pixels of given picture be represented in \(L\) gray levels \([1,2,\ldots,L]\). The number of pixels with level \(i\) is denoted by \(n_i\) and the total number of pixels by \(N = n_1 + n_2 + \ldots + n_L\).

In order to simplify the discussion, the gray-level histogram is normalized and regarded as a probability distribution:

\[
p_i = \frac{n_i}{N},
\]

\[
\sum_{i=0}^{L} p_i = 1
\]

Now suppose that we dichotomize the pixel into two classes \(C_0\) and \(C_1\) (foreground and objects) by a threshold at level \(k\): \(C^0\) denotes pixels with levels \([0, \ldots k]\) and \(C^1\) denotes pixels with levels \([k+1, \ldots ,L]\). Then the probabilities of class occurrence and the class mean levels, respectively, are given by
Image binarization. The Otsu method

Written by Simeon Antonov - Last Updated Monday, 31 May 2010 08:35

\[ \omega_0 = \Pr(C_0) = \sum_{i=0}^{k} p_i = \omega(k) \]

\[ \omega_1 = \Pr(C_1) = \sum_{i=k+1}^{L} p_i = 1 - \omega(k) \]

\[ \mu_0 = \sum_{i=0}^{k} i \Pr(i \mid C_0) = \mu(k) / \omega(k) \]

\[ \mu_1 = \sum_{i=k+1}^{L} i \Pr(i \mid C_1) = (\mu_T - \mu(k)) / (1 - \omega(k)) \]

where

\[ \mu(k) = \sum_{i=0}^{k} ip_i \quad \mu_T = \sum_{i=0}^{L} ip_i \]

The optimal threshold is

\[ \sigma_B^2(k^*) = \max_{0 \leq k \leq L} \sigma_B^2(k) \]

where

\[ \sigma_B^2(k) = \frac{[\mu_T \omega(k) - \mu(k)]^2}{\omega(k)[1-\omega(k)]} \]

\[ + \frac{\omega(k)[1-\omega(k)]}{\omega(k)[1-\omega(k)]} \]

Reference:
1. Nobuyuki Otsu, A threshold selection method from gray level histograms, IEEE, January 1979

```cpp
void OtsuBinarization(Image* input_image, Image* output_image)
{
    int max_x = input_image->get_width();
    int max_y = input_image->get_height();
    const int L = 256;
    float hist[L]={0.0F};
    //calculate grayscale histogram
    for (int x=0; x < max_x; ++x)
        for(int y=0; y < max_y; ++y)
        {
            Pixel cur;
            cur=input_image->get_pixel(x, y);
            int graylevel = max(0.0, min(255.0, 0.299*cur.R + 0.587*cur.G + 0.114*cur.B));
            hist[graylevel]+=1;
        }
    int N = max_x*max_y;
    //normalize histogram
    for (int i=0; i < L; ++i)
        hist[i]/=N;
    float ut = 0;
    for (int i=0; i < L; ++i)
        ut+=i*hist[i];
    int max_k=0;
    int max_sigma_k_=0;
    for (int k=0; k < L;++k)
    {
        float wk = 0;
        for (int i = 0; i <=k;++i)
            wk += hist[i];
        float uk = 0;
        for (int i = 0; i <=k;++i)
            uk+= i*hist[i];
        float sigma_k = 0;
        if (wk !=0 && wk!=1)
            sigma_k = ((ut*wk - uk)*(ut*wk - uk))/(wk*(1-wk));
        if (sigma_k > max_sigma_k_)
        {
            max_k = k;
            max_sigma_k_ = sigma_k;
        }
    }
    for (int x =0; x < max_x;++x)
        for (int y =0; y < max_y;
            {
            Pixel cur;
            cur=input_image->get_pixel(x,y);
            int graylevel = max(0.0, min(255.0, 0.299*cur.R + 0.587*cur.G + 0.114*cur.B));
            if (graylevel < max_k)
                output_image->set_pixel(x,y, RGB(0, 0, 0));
            else
                output_image->set_pixel(x,y, RGB(255, 255, 255));
        }
    }
```