OpenCV--图像处理

接下来细说一下关于OpenCV提供的图片处理函数方法。

- 平滑处理

平滑处理又称为模糊处理，是一项使用频率很高的图像处理方法。平滑处理的用途有很多，最常见的是用来减少图像上的噪点和失真，降低图像的分辨率时，平滑处理是非常重要的手段。OpenCV提供了cvSmooth方法和cvErode方法用于图像的平滑和融合。

```
/** Image smooth methods */
enum SmoothMethod_c
{
    /** linear convolution with \f$\texttt{size1}\times\texttt{size2}\f$ box kernel
(all 1's). If
    you want to smooth different pixels with different-size box kernels, you can use
the integral
    image that is computed using integral */
    CV_BLUR_NO_SCALE =0,
    /** linear convolution with \f$\texttt{size1}\times\texttt{size2}\f$ box kernel
(all
    1's) with subsequent scaling by \f$1/(\texttt{size1}\cdot\texttt{size2})\f$ */
    CV_BLUR  =1,
    /** linear convolution with a \f$\texttt{size1}\times\texttt{size2}\f$ Gaussian
kernel */
    CV_GAUSSIAN  =2,
    /** median filter with a \f$\texttt{size1}\times\texttt{size1}\f$ square aperture
*/
    CV_MEDIAN =3,
    /** bilateral filter with a \f$\texttt{size1}\times\texttt{size1}\f$ square
aperture, color
    sigma= sigma1 and spatial sigma= sigma2. If size1=0, the aperture square side is
set to
    cvRound(sigma2\*1.5)\*2+1. See cv::bilateralFilter */
    CV_BILATERAL =4
};

/** @brief Smooths the image in one of several ways.
@param src The source image
@param dst The destination image
@param smoothtype Type of the smoothing, see SmoothMethod_c
@param size1 The first parameter of the smoothing operation, the aperture width. Must
be a
positive odd number (1, 3, 5, ...)
@param size2 The second parameter of the smoothing operation, the aperture height.
Ignored by
CV_MEDIAN and CV_BILATERAL methods. In the case of simple scaled/non-scaled and
Gaussian blur if
size2 is zero, it is set to size1. Otherwise it must be a positive odd number.
@param sigma1 In the case of a Gaussian parameter this parameter may specify Gaussian
\f$\sigma\f$
(standard deviation). If it is zero, it is calculated from the kernel size:
\f[\sigma  = 0.3 (n/2 - 1) + 0.8  \quad   \text{where}   \quad  n= \begin{array}{l l}
\mbox{\texttt{size1} for horizontal kernel} \\ \mbox{\texttt{size2} for vertical
kernel} \end{array}\f]
Using standard sigma for small kernels ( \f$3\times 3\f$ to \f$7\times 7\f$ ) gives
```

```
better speed. If
sigma1 is not zero, while size1 and size2 are zeros, the kernel size is calculated from
the
sigma (to provide accurate enough operation).
@param sigma2 additional parameter for bilateral filtering
@see cv::GaussianBlur, cv::blur, cv::medianBlur, cv::bilateralFilter.
*/

CVAPI(void) cvSmooth( const CvArr* src, CvArr* dst,
                      int smoothtype CV_DEFAULT(CV_GAUSSIAN),
                      int size1 CV_DEFAULT(3),
                      int size2 CV_DEFAULT(0),
                      double sigma1 CV_DEFAULT(0),
                      double sigma2 CV_DEFAULT(0));
```

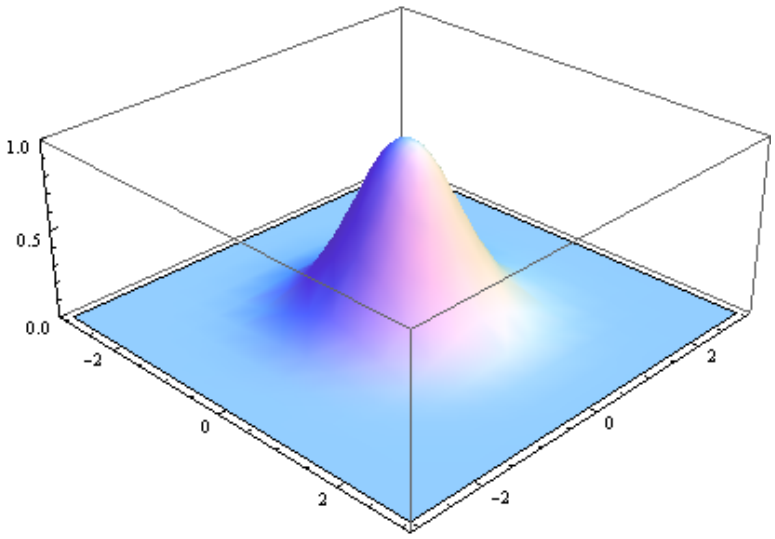
cvSmooth方法提供了几种调用方式，依据smoothType来限定平滑的效果。函数方法默认使用了GAUSSIAN的方式来处理。函数包括了4个参数，分别代表如下：

平滑类型	名称	支持	输入类型	输出类型	说明
CV_BLUR	简单模糊	是	8u 32f	8u 32f	对每个像素做，aram1*param2和，并缩放 1/(ma1* sigma2)
CV_BLUR_NO_SCALE	简单无缩放模糊	否	8u		对每个像素做，aram1*param2和
CV_median	中值滤波	否	8u	8u	对每个像素做，aram1*param2止滤波
CV_gaussian	高斯滤波	是	8u 32f	8u 32f	对每个像素做，aram1*param2高斯卷积
CV_BILATERAL	双边滤波	否	8u	8u	双线性3*3滤波颜色sigma1-pam1 空间sigmaaram2

高斯平滑滤波的零均值函数如下：

$$g(x)=exp(-x^2/(2 sigma^2)$$

高斯滤波需要高斯核和图像做卷积，openCV的参数可以是几个基本的卷积形式，比如3*3 7*7等。依据公式推导出满足正态分布的高斯核，越靠近中心点的，越模糊。



、 3*3高斯核推导：

0.0947416	0.118318	0.0947416
0.118318	0.147761	0.118318
0.0947416	0.118318	0.0947416

利用此高斯核与矩阵进行卷积计算。得出最后滤波的结果图像。
高斯核计算公式：

$$G(r)=\frac{1}{\sqrt{2\pi\sigma^2}^{N}}e^{-r^2/(2\sigma^2)} \quad (1\text{-}1)$$

二维模板计算公式：

$$G(x,y)=\frac{1}{2\pi\sigma^2}e^{-\frac{(x-m/2)^2+(y-n/2)^2}{2\sigma^2}} \quad (1\text{-}2)$$

C实现代码历程：

```
#define  PI 3.141592653
```

```
double* GaussianSmooth2D(double sigma)
{
    //确保sigma为正数
    sigma = sigma > 0 ? sigma : 0;
    //高斯核矩阵的大小为(6*sigma+1)*(6*sigma+1)
    //ksize为奇数
    int ksize = round(sigma * 3) * 2 + 1;
    if(ksize == 1)
    {
        return NULL;
    }
    //计算高斯核矩阵
    cout<< "ksize "<<ksize<<endl;
    double *kernel = new double[ksize*ksize];
    double scale = -0.5/(sigma*sigma);
    double cons = -scale/PI;
    double sum = 0;
    for(int i = 0; i < ksize; i++)
    {
        for(int j = 0; j < ksize; j++)
        {
            int x = i-(ksize-1)/2;
            int y = j-(ksize-1)/2;
            kernel[i*ksize + j] = cons * exp(scale * (x*x + y*y));
            sum += kernel[i*ksize+j];
        }
    }
    //归一化
    for(int i = ksize*ksize-1; i >=0; i--)
    {
        *(kernel+i) /= sum;
    }
    return kernel;
}
```

原理二维卷积算法：

```
static double* gauss2d(long radius)
{
    radius = radius > 0 ? radius : 0;
    int ksize = round(radius * 3) * 2 + 1;
    if (ksize == 1)
    {
        return NULL;
    }
    //计算高斯核矩阵
    double* kernel = new double[ksize * ksize];
    static const double SQRT2PI = sqrt(2.0 * PI);
    double              sigma   = (double) radius / 3.0;
    double              sigma2  = 2.0 * sigma * sigma;
    double              sigmap  = sigma * SQRT2PI;
    double              sum     = 0;
    for (long           n       = 0, i = -radius; i <= radius; ++i, ++n)
    {
        kernel[n] = exp(-(double) (i * i) / sigma2) / sigmap;
        sum += kernel[n];
    }
    for (long           n       = 0, i = -radius; i <= radius; ++i, ++n)
    {
```

```
        kernel[n] /= sum;
    }
    return kernel;
}
static void gaussBlurSouce(argb* pix, int w, int h, int radius)
{
    if (pix == NULL || w < 0 || h < 0 || radius < 0)
    {
        return;
    }
    int ksize = round(radius * 3) * 2 + 1;
    double* kernel = gaussianSmooth2D(radius);
    double sumR = 0, sumG = 0, sumB = 0;
    argb* out = (argb*) malloc(sizeof(argb) * w * h);
    int      kradius = ksize / 2;
    for (int y       = 0; y < h; ++y)
    {
        for (int x = 0; x < w; ++x)
        {
            sumR = sumG = sumB = 0;
            if (y == 0 || y == h - 1)
            {
                sumR = sumG = sumB = 0;
            } else if (x == 0 || x == w - 1)
            {
                sumR = sumG = sumB = 0;
            } else
            {
                for (int i = -kradius; i < kradius; ++i)
                {
                    long i_k = y + i < h ? y + i : h;
                    i_k = y + i > 0 ? y + i : 0;
                    for (int j = -kradius; j < kradius; ++j)
                    {
                        long j_k = x + i < w ? x + i : w;
                        j_k      = x + i > 0 ? x + i : 0;
                        long inx_k  = i_k * w + j_k;
                        int  valueR = (pix[inx_k]).red;
                        int  valueG = (pix[inx_k]).green;
                        int  valueB = (pix[inx_k]).blue;
                        sumR += kernel[(i + kradius) * kradius + j + kradius] * valueR;
                        sumG += kernel[(i + kradius) * kradius + j + kradius] * valueG;
                        sumB += kernel[(i + kradius) * kradius + j + kradius] * valueB;
                    }
                }
                sumR       = (int) abs(sumR);
                sumG       = (int) abs(sumG);
                sumB       = (int) abs(sumB);
            }
            sumR = (uint8_t) sumR > 0xff ? 0xff : sumR;
            sumG = (uint8_t) sumG > 0xff ? 0xff : sumG;
            sumB = (uint8_t) sumB > 0xff ? 0xff : sumB;
            argb argb1;
            argb1.blue  = sumB;
            argb1.red   = sumR;
            argb1.green = sumG;
            argb1.alpha = pix[x + y * w].alpha;
            out[x + y * w] = argb1;
        }
```

```
    }
    for (int i       = 0; i < h; ++i)
    {
        for (int j = 0; j < w; ++j)
        {
            pix[j + i * w] = out[j + i * w];
        }
    }
    free(out);
}
```

改进纵横卷积算法：

```
static double* gauss2d(long radius)
{
    radius = radius > 0 ? radius : 0;
    int ksize = round(radius * 3) * 2 + 1;
    if (ksize == 1)
    {
        return NULL;
    }
    //计算高斯核矩阵
    double* kernel = new double[ksize * ksize];
    static const double SQRT2PI = sqrt(2.0 * PI);
    double              sigma   = (double) radius / 3.0;
    double              sigma2  = 2.0 * sigma * sigma;
    double              sigmap  = sigma * SQRT2PI;
    double              sum     = 0;
    for (long          n       = 0, i = -radius; i <= radius; ++i, ++n)
    {
        kernel[n] = exp(-(double) (i * i) / sigma2) / sigmap;
        sum += kernel[n];
    }
    for (long          n       = 0, i = -radius; i <= radius; ++i, ++n)
    {
        kernel[n] /= sum;
    }
    return kernel;
}
static void gaussblur2d(argb* pix, int w, int h, int radius)
{
    if (pix == NULL || w < 0 || h < 0 || radius < 0)
    {
        return;
    }
    double* kernel = gauss2d(radius);
    argb  * outW   = (argb*) malloc(sizeof(argb) * w * h);
    argb  * outH   = (argb*) malloc(sizeof(argb) * w * h);
    memset(outW, 0, sizeof(argb) * w * h);
    memset(outH, 0, sizeof(argb) * w * h);
    int    kradius = radius;
    double sumR    = 0, sumG = 0, sumB = 0;
    for (int y = 0; y < h; ++y)
    {
        for (int x = 0; x < w; ++x)
        {
            sumB = sumG = sumR = 0;
            if (y == 0 || y == h - 1)
            {
```

```
                continue;
            } else if (x == 0 || x == w - 1)
            {
                continue;
            }
            for (long n = 0, i = -kradius; i <= kradius; ++i, ++n)
            {
                long i_k = x + i < w ? x + i : w;
                i_k = x + i > 0 ? x + i : 0;
                int inx_k = i_k + y * w;
                sumR += pix[inx_k].red * kernel[n];
                sumG += pix[inx_k].green * kernel[n];
                sumB += pix[inx_k].blue * kernel[n];
            }
            sumR        = (uint8_t) sumR > 0xff ? 0xff : sumR;
            sumG        = (uint8_t) sumG > 0xff ? 0xff : sumG;
            sumB        = (uint8_t) sumB > 0xff ? 0xff : sumB;
            argb argb1;
            argb1.blue  = sumB;
            argb1.red   = sumR;
            argb1.green = sumG;
            argb1.alpha = pix[x + y * w].alpha;
            outW[x + y * w] = argb1;
        }
    }
    for (int x = 0; x < w; ++x)
    {
        for (int y = 0; y < h; ++y)
        {
            if (y == 0 || y == h - 1)
            {
                continue;
            }
            if (x == 0 || x == w - 1)
            {
                continue;
            }
            sumB = sumG = sumR = 0;
            for (long n = 0, i = -kradius; i <= kradius; ++i, ++n)
            {
                long i_k = y + i < h ? y + i : h;
                i_k      = y + i > 0 ? y + i : 0;
                long inx_k = x + i_k * w;
                sumR += pix[inx_k].red * kernel[n];
                sumG += pix[inx_k].green * kernel[n];
                sumB += pix[inx_k].blue * kernel[n];
            }
            sumR        = (uint8_t) sumR > 0xff ? 0xff : sumR;
            sumG        = (uint8_t) sumG > 0xff ? 0xff : sumG;
            sumB        = (uint8_t) sumB > 0xff ? 0xff : sumB;
            argb argb1 = outW[x + y * w];
            argb1.blue  = (sumB + argb1.blue) / 2;
            argb1.red   = (sumR + argb1.red) / 2;
            argb1.green = (sumG + argb1.green) / 2;
            argb1.alpha = pix[x + y * w].alpha;
            outH[x + y * w] = argb1;
        }
    }
    for (int i = 1; i < h - 1; ++i)
```

```
    {
        for (int j = 1; j < w - 1; ++j)
        {
            pix[j + i * w] = outH[j + i * w];
        }
    }
    delete (outW);
    delete (outH);
}
```

双边滤波含有两个参数，第一个代表了空域中所使用的高斯核高度，第二个参数代表了高斯核的宽度，第二个参数越大，表示滤波范围越大。

- 图像形态学

图像形态学变化，openCV提供了快速、便捷的函数方法。基本的形态包括了图形的腐蚀、膨胀等功能。例如分割图形，消除噪声等。

1、膨胀与腐蚀

膨胀是指将一些图片与对应的核进行卷积。核可以是任何的大小或者形状，它具有一个单独定义出来的参考点。多数情况下，核是一个带有参考点的实心正方形或者圆盘。膨胀是求局部最大值的操作，核B和图像卷积，计算出在B覆盖的区域中像素点的最大值。并把这个最大值赋值给参考点的像素值。

腐蚀是膨胀的反操作。服饰是需要计算区域中像素的最小值。当计算完成时，最小的点将被放到对应的像素位置上。

我们可以依据cvErode和cvDilate来进行上述的操作。

```
/** @brief erodes input image (applies minimum filter) one or more times.
   If element pointer is NULL, 3x3 rectangular element is used
@see cv::erode
*/
CVAPI(void)  cvErode( const CvArr* src, CvArr* dst,
                      IplConvKernel* element CV_DEFAULT(NULL),
                      int iterations CV_DEFAULT(1) );
/** @brief dilates input image (applies maximum filter) one or more times.
   If element pointer is NULL, 3x3 rectangular element is used
@see cv::dilate
*/
CVAPI(void)  cvDilate( const CvArr* src, CvArr* dst,
                       IplConvKernel* element CV_DEFAULT(NULL),
                       int iterations CV_DEFAULT(1) );
```

他们都会支持in-place操作，原图像和目标图像是同一个图像。第三个参数是对应的核算子。openCV中，调用cvErode方法，将某点p的像素值设置为p对应核覆盖下所有点的最小值，同样的，使用膨胀时，取得是最大值。对应计算公式:

```
    dst=erode(src,element):  dst(x,y)=min((x',y') in element))src(x+x',y+y')
    dst=dilate(src,element): dst(x,y)=max((x',y') in element))src(x+x',y+y')
```

自定义核。

openCV提供了自定义IplConvKernel的一种核。cvCreateStructuringElementEx方法来创建一个卷积核。

```
/** Shapes of a structuring element for morphological operations
```

```
@see cv::MorphShapes, cv::getStructuringElement
*/
enum MorphShapes_c
{
    CV_SHAPE_RECT      =0,
    CV_SHAPE_CROSS     =1,
    CV_SHAPE_ELLIPSE   =2,
    CV_SHAPE_CUSTOM    =100 //!< custom structuring element
};

/** @brief Returns a structuring element of the specified size and shape for
morphological operations.
@note the created structuring element IplConvKernel\* element must be released in the
end using
`cvReleaseStructuringElement(&element)`.
@param cols Width of the structuring element
@param rows Height of the structuring element
@param anchor_x x-coordinate of the anchor
@param anchor_y y-coordinate of the anchor
@param shape element shape that could be one of the cv::MorphShapes_c
@param values integer array of cols*rows elements that specifies the custom shape of
the
structuring element, when shape=CV_SHAPE_CUSTOM.
@see cv::getStructuringElement
*/

CVAPI(IplConvKernel*)  cvCreateStructuringElementEx(
            int cols, int  rows, int  anchor_x, int  anchor_y,
            int shape, int* values CV_DEFAULT(NULL) );

/** @brief releases structuring element
@see cvCreateStructuringElementEx
*/
CVAPI(void)  cvReleaseStructuringElement( IplConvKernel** element );
```

形态核与卷积核不同，形态核不需要填充数据。shape参数为指定一个对应的形态核。默认是一个所有值为非空的矩形核。

在处理布尔图片或者图像掩码时，基本的腐蚀和膨胀操作是可以满足图像分割需求的。然而，在处理灰度或者彩色图像时，往往需要一些额外的操作。更通用的是cvMorphologyEx方法。

```
/** Morphological operations */
enum
{
    CV_MOP_ERODE        =0,
    CV_MOP_DILATE       =1,
    CV_MOP_OPEN         =2,
    CV_MOP_CLOSE        =3,
    CV_MOP_GRADIENT     =4,
    CV_MOP_TOPHAT       =5,
    CV_MOP_BLACKHAT     =6
};

/** @brief Performs complex morphological transformation
@see cv::morphologyEx
*/

CVAPI(void)  cvMorphologyEx( const CvArr* src, CvArr* dst,
                             CvArr* temp, IplConvKernel* element,
                             int operation, int iterations CV_DEFAULT(1) );
```

- 开运算和闭运算

开运算和闭运算包含了图像形态学操作的腐蚀和膨胀操作，在开运算的图像中，我们首先对图像进行了腐蚀操作，然后对图像进行膨胀操作。开运算通常是对图像的区域化进行统计，常用与二值化图像。开运算可以使两个区块分割，然后再统计区块的数量。闭运算时，我们先膨胀然后在腐蚀，闭运算可以消除噪声引起的干扰，通常我们都是先闭运算消去干扰，然后在开运算分割图像，连接图像的联通区域。

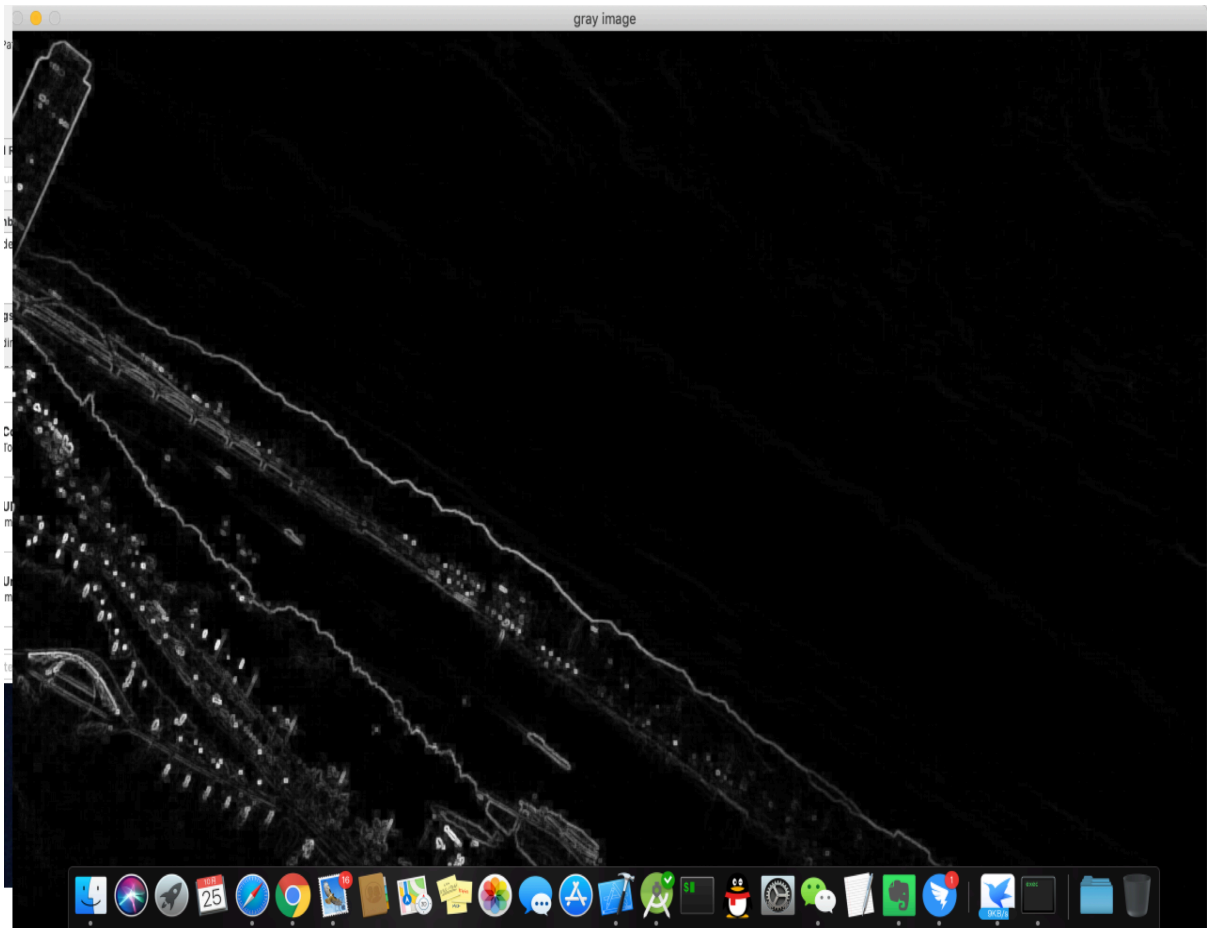
开运算和闭运算原理可以参考博客：
https://www.cnblogs.com/daxiongblog/p/6289551.html

- 形态学梯度

形态学梯度按照公式：
gradlent(src) = dilate(src) - erode(src)
对一副二值化图片进行上述操作，可以使得边缘突出来。利用这个方式，可以知道图像的亮度变化的剧烈程度。测试历程代码：

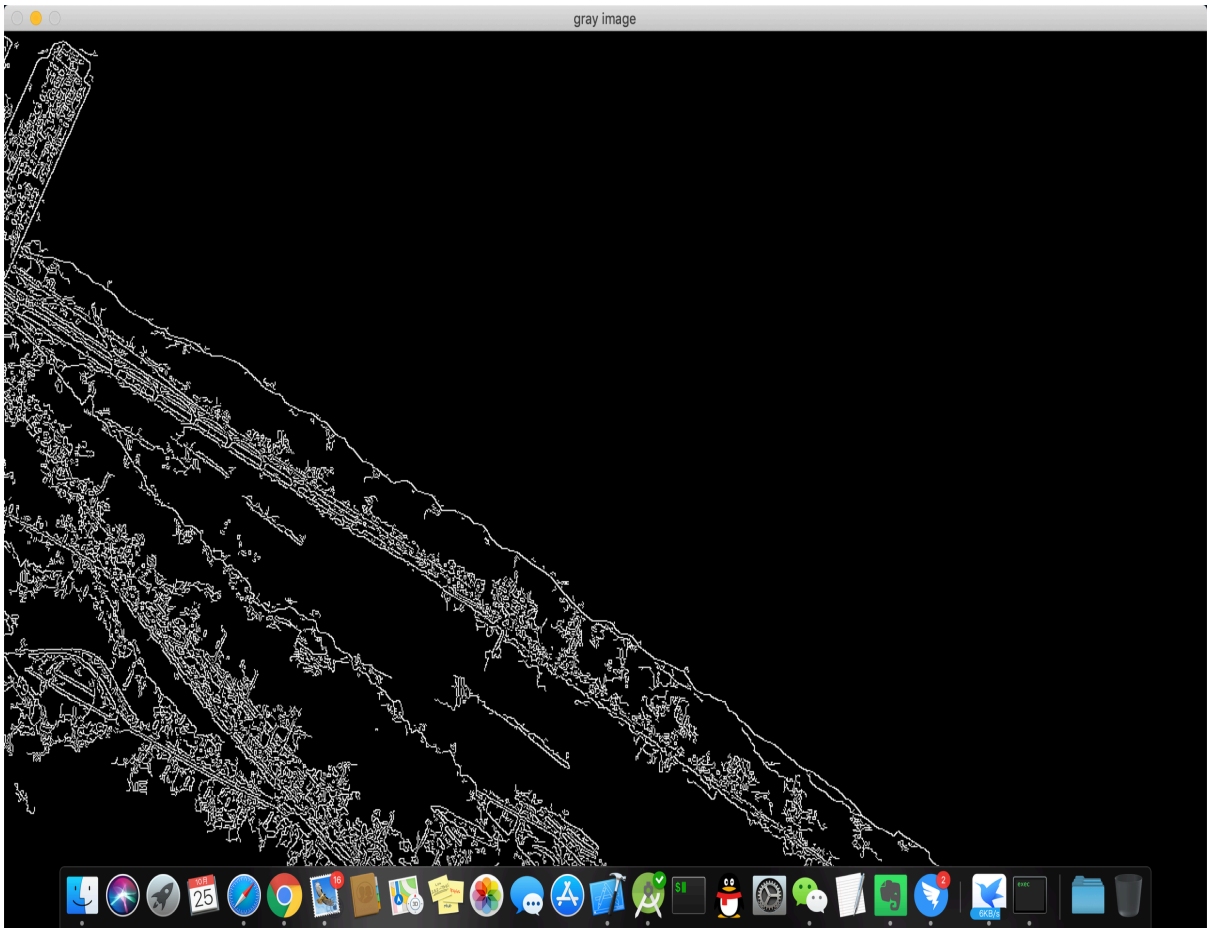
```
//
//  main.cpp
//  Erode
//
//  Created by 凌志 on 2018/10/24.
//  Copyright © 2018 凌志. All rights reserved.
//
#include <iostream>
#include "opencv/cv.h"
#include "opencv/highgui.h"
using namespace std;
using namespace cv;
#define IMAGE_LOAD_PATH  "/Users/genesis/Pictures/IMG_3393.JPG"
int main(int argc, const char * argv[]) {
    IplImage *input = cvLoadImage(IMAGE_LOAD_PATH,CV_LOAD_IMAGE_GRAYSCALE);
    IplImage *dilate = cvCreateImage(cvSize(input->width, input->height), input->depth,
input->nChannels);
    IplImage *erode = cvCreateImage(cvSize(input->width, input->height), input->depth,
input->nChannels);
    cvDilate(input, dilate);
    cvErode(input, erode);
    cvSub(dilate, erode, input);
    cvShowImage("gray image", input);
    cvWaitKey();
    cvReleaseImage(&input);
    cvReleaseImage(&dilate);
    cvReleaseImage(&erode);
    return 0;
}
```

实际测试效果：

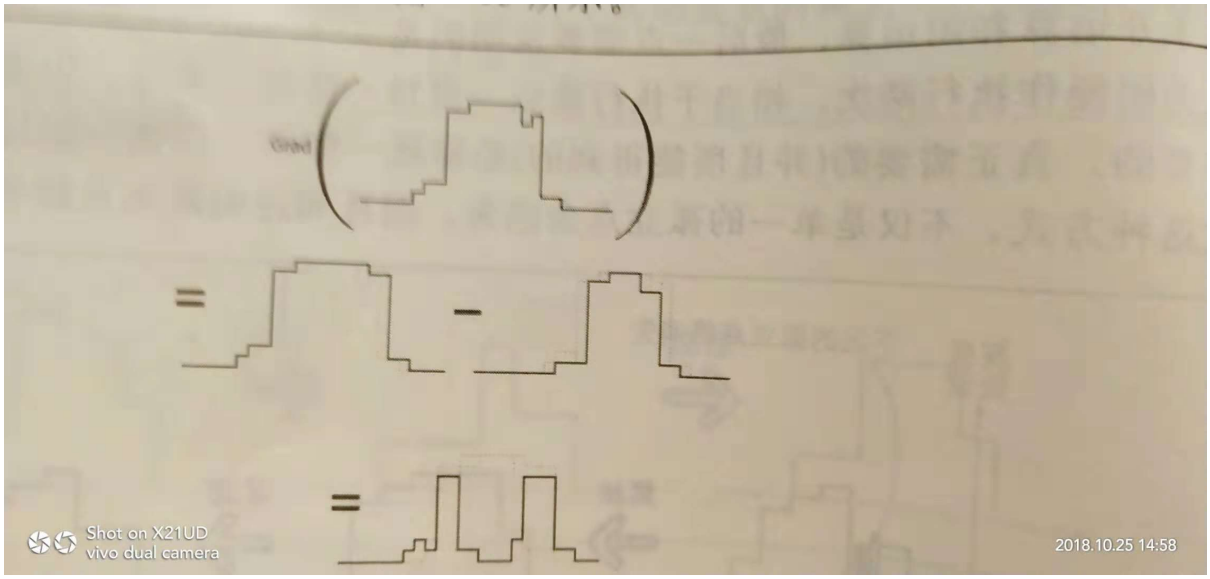


实际效果，先对图片进行膨胀操作，然后对图片进行腐蚀操作，最后按照公式，得到了一张效果不错的边缘图。

相对于一般的边缘检测，例如Canny算子，效果滤波更干净。



形态学梯度原理解释：



- 礼帽和黑帽

最后两个操作被称为”礼帽“(Top hat)和”黑帽“ (Black hat)，这些操作用于分离比邻相近的亮点或者暗色块。当需要孤立相对近的部分，可以采用这样的方式。这两个方式的公式如下：

TopHat(src) = src - open(src)

BlackHat (src) = close (src) - src

礼帽的操作是用A减去A的开运算，开运算的结果是降低局部亮度，因此，礼帽可以突出周围更鲜明的部分。同理对比black hat，可以突出A周围更黑暗的部分。

- 浸水填充算法

浸水填充是一个非常有效的功能，它经常用于标记或者分立图像的一部分以便于对其进行进一步处理和分析，浸水填充也可以用来从输入图像获取掩码区域，掩码会加速处理过程，或只处理掩码指定的像素点。cvFloodFill函数本身就包含了一个可选择的掩码参数，用来进一步控制这些区域将被填充的颜色。

浸水填充是填充算法中最为常见的一种，对于同类填充算法，都必须要选择一个种子点，然后把相邻的两个像素值染成同一个颜色，浸水填充操作的结果总是某个联通的区域，当相近的像素点接近指定范围时，cvFloodFill方法会为这个点图上颜色。方法定义：

```
/** @br
/** FloodFill flags */
enum
{
    CV_FLOODFILL_FIXED_RANGE =(1 << 16),
    CV_FLOODFILL_MASK_ONLY   =(1 << 17)
};

ief Fills the connected component until the color difference gets large enough
@see cv::floodFill
*/
CVAPI(void)  cvFloodFill( CvArr* image, CvPoint seed_point,
                          CvScalar new_val, CvScalar lo_diff
CV_DEFAULT(cvScalarAll(0)),
                          CvScalar up_diff CV_DEFAULT(cvScalarAll(0)),
                          CvConnectedComp* comp CV_DEFAULT(NULL),
                          int flags CV_DEFAULT(4),
                          CvArr* mask CV_DEFAULT(NULL));
```

浸水填充从对应点seedpoint 开始，如果一个像素点不低于被染色的相邻点减去lodiff且不高于相邻点加上updiff，那么该像素就会被染色。如果flag参数报告韩了CV_FLOODFILL_FIXED_RANGE，这时每个像素点都会和这个种子点进行比较，而不是相邻的两个点比较。并且不会覆盖mask中非零的点。

同时，这个flag是一个高位和低位分开表示的方法。一共是4字节。

- 尺寸调整

我们经常会将某种尺寸的图像调整为另一尺寸的图像，比如放大和缩小，我们可以用cvResize函数来放大或者缩小图片。如果原图像中设置了ROI，那么cvResize会对ROI区域调整尺寸，以匹配目标图像。如果目标图像已经设置了ROI值，那么会依据尺寸填充到目标图像的ROI中。

```
/** Sub-pixel interpolation methods */
enum
{
    CV_INTER_NN        =0,        //临近差值
    CV_INTER_LINEAR    =1,        //线性插值
    CV_INTER_CUBIC     =2,        //三次样条插值
    CV_INTER_AREA      =3,        //区域插值
    CV_INTER_LANCZOS4  =4         //LANCZ插值算法
};

/** @brief Resizes image (input array is resized to fit the destination array)
@see cv::resize
*/
```

```
CVAPI(void)  cvResize( const CvArr* src, CvArr* dst,
                       int interpolation CV_DEFAULT( CV_INTER_LINEAR ));
```

cvResize方法最后一个参数代表了resize所使用的方法，具体的类型如上所示的插值方式。依据输入不同，插值算法也有所不同。

- 图像金字塔

图像金字塔是一个图像集合体，集合中所有的图像都源自于同一个原始图像，而且是通过原始图像的连续采样活的，直到达到某个终止条件才会停止采样。

有两种类型的图像金字塔常常出现在文献和应用中，高斯金字塔和拉普拉斯金字塔。高斯金字塔采用了向下降图像采样频率的方式，而拉普拉斯金字塔是从低层图像向上采样重建图像的方式。要从金字塔i层生成i+1层（我们用G（i+1）表示i+1层），首先我们需要用高斯核对G（i）进行图像卷积。然后删除对应图像的偶数列和偶数行，得到的图像面积会变为原始图像的1/4，按照上述的操作对图像循环操作，可以产生整个金字塔。

```
/** @brief Smoothes the input image with gaussian kernel and then down-samples it.
   dst_width = floor(src_width/2)[+1],
   dst_height = floor(src_height/2)[+1]
   @see cv::pyrDown
*/
CVAPI(void)  cvPyrDown( const CvArr* src, CvArr* dst,
                        int filter CV_DEFAULT(CV_GAUSSIAN_5x5) );
/** @brief Up-samples image and smoothes the result with gaussian kernel.
   dst_width = src_width*2,
   dst_height = src_height*2
   @see cv::pyrUp
*/
CVAPI(void)  cvPyrUp( const CvArr* src, CvArr* dst,
                      int filter CV_DEFAULT(CV_GAUSSIAN_5x5) );
```

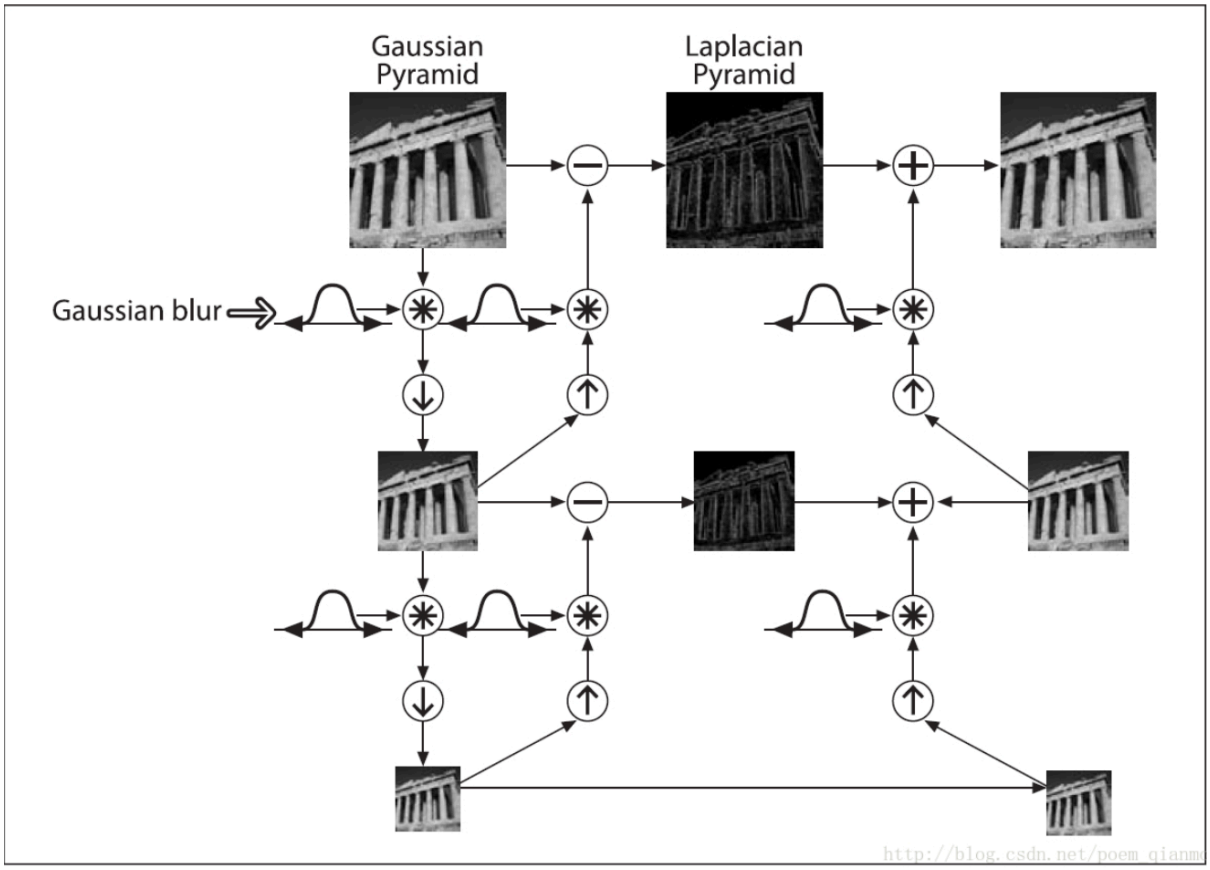
cvPyrUp可以在原有的图像基础上，把图像的每个维度都扩大到原先的两倍，在指定的滤波器进行卷积运算后，可以得到丢失像素的近似值。pyrDown并不是pyrUP的逆操作，之所以这样是因为pyrDown是一个丢失信号方法，为了恢复原来的图像，我们需要获取由降采样操作丢失的信息。这些数据形成了拉普拉斯金字塔。下面是拉普拉斯金字塔第i层的数学定义：

$$L(i)=G(i)-UP(G(i+1)\bigotimes)G(5*5)$$

这里的Up操作是将原本的（x,y）映射到目标图像的（2x+1,2y+1）的位置 $G(5*5)$代表了高斯5*5卷积核。我们也可以依据这个特性，直接用cvPyrUp方法进行拉普拉斯卷积运算：

$$L(i)=G(i)-UP(G(i+1)$$

算法原理：



金字塔操作可以利用算法对图像进行图像分割，在低分辨率的情况下，逐级对图像分层优化分割图像效果。opencv提供了cvPyrSegmentation方法来实现这一效果 。

- 阈值化

openCV提供了cvThreshold方法来实现阈值化。

```
/** Threshold types */
enum
{
    CV_THRESH_BINARY      =0,  /**< value = value > threshold ? max_value : 0       */
    CV_THRESH_BINARY_INV  =1,  /**< value = value > threshold ? 0 : max_value       */
    CV_THRESH_TRUNC       =2,  /**< value = value > threshold ? threshold : value   */
    CV_THRESH_TOZERO      =3,  /**< value = value > threshold ? value : 0           */
    CV_THRESH_TOZERO_INV  =4,  /**< value = value > threshold ? 0 : value           */
    CV_THRESH_MASK        =7,
    CV_THRESH_OTSU        =8, /**< use Otsu algorithm to choose the optimal threshold
value;
                                  combine the flag with one of the above CV_THRESH_*
values */
    CV_THRESH_TRIANGLE    =16  /**< use Triangle algorithm to choose the optimal
threshold value;
                                  combine the flag with one of the above CV_THRESH_*
values, but not
                                  with CV_THRESH_OTSU */
};

/** @brief Applies fixed-level threshold to grayscale image.
   This is a basic operation applied before retrieving contours
```

```
@see cv::threshold
*/
CVAPI(double)  cvThreshold( const CvArr*  src, CvArr*  dst,
                            double  threshold, double  max_value,
                            int threshold_type );
```

- 自适应阈值化

这是一种改进型的阈值化操作，openCV提供了cvAdaptIveThreshold方法。

```
/** Adaptive threshold methods */
enum
{
    CV_ADAPTIVE_THRESH_MEAN_C  =0,
    CV_ADAPTIVE_THRESH_GAUSSIAN_C  =1
};
/** @brief Applies adaptive threshold to grayscale image.
   The two parameters for methods CV_ADAPTIVE_THRESH_MEAN_C and
   CV_ADAPTIVE_THRESH_GAUSSIAN_C are:
   neighborhood size (3, 5, 7 etc.),
   and a constant subtracted from mean (...,-3,-2,-1,0,1,2,3,...)
@see cv::adaptiveThreshold
*/
CVAPI(void)  cvAdaptiveThreshold( const CvArr* src, CvArr* dst, double max_value,
                                  int adaptive_method
CV_DEFAULT(CV_ADAPTIVE_THRESH_MEAN_C),
                                  int threshold_type CV_DEFAULT(CV_THRESH_BINARY),
                                  int block_size CV_DEFAULT(3),
                                  double param1 CV_DEFAULT(5));
```

自适应阈值法是通过周围bxb区域的像素的加权平均，然后减去一个常数来得到自适应阈值。b有参数blocksize来确定，选择第一个参数是对区域内的像素平均加权，选择第二个参数，按照高斯函数离中心的点加权计算。