OpenCV--图像变换

- 卷积

卷积是本章所讨论的基础，从抽象层次上说，这个意味着需要对图像的每一个部分进行操作。一个特殊卷积所实现的功能是由其卷积核决定的。这个核本质是一个固定大小，由数值参数构成的数组，数组的参考点通常位于数组的中心。数组的大小称为核支撑。单就技术而言，核支撑实际上仅仅由核数组的非0部分组成。对于每一个核点，我们可以得到这个点的核值以及图像中和对应像素点的值。将这些值相乘并求和，并将这个结果放在输入图像参考点相应的位置上，通过整幅图片扫描卷积核，对图像的每个像素进行操作。代表公式如下：

设矩阵 A 和矩阵 B， A 的行数和列数分别为 Mr, Mc， B 的行数和列数分别为 Nr, Nc,

则有：

$$C(s,t)=\sum_{m=0}^{Mr-1}\sum_{n=0}^{Mc-1}A(m,n)\cdot B(s-m,t-n)$$

可以使用OpenCV提供的优化方法，cvFilte2D方法。

```
/** @brief Convolves an image with the kernel.
@param src input image.
@param dst output image of the same size and the same number of channels as
src.
@param kernel convolution kernel (or rather a correlation kernel), a
single-channel floating point
matrix; if you want to apply different kernels to different channels, split
the image into
separate color planes using split and process them individually.
@param anchor anchor of the kernel that indicates the relative position of
a filtered point within
the kernel; the anchor should lie within the kernel; default value (-1,-1)
means that the anchor
is at the kernel center.
@see cv::filter2D
*/

CVAPI(void) cvFilter2D( const CvArr* src, CvArr* dst, const CvMat* kernel,
                        CvPoint anchor CV_DEFAULT(cvPoint(-1,-1)));
```

在默认情况下，进行卷积操作之前，openCV会复制目标图像的边界创建虚拟像素，这样便于目标图像dst边界的像素可以被填充。复制是通过了input(-dx.y) =input(0,y), input（w+dx,y）= input （w-1,y）的方式来实现的。顺道注意下，卷积核是32F的数据类型。

- 卷积边界

对于卷积，一个很自然的问题是如何处理卷积边界。许多使用cvFilter2D方法的需要使用内部各类方法来处理，同样，在卷积运算中，需要知道如何高效地解决这个问题。openCV提供了cvCopyMakeBorder方法。

```
/*************************************************************************
*************\
*                                    Image
Processing                                          *
\*************************************************************************
*************/
/** Copies source 2D array inside of the larger destination array and
   makes a border of the specified type (IPL_BORDER_*) around the copied
area. */

CVAPI(void) cvCopyMakeBorder( const CvArr* src, CvArr* dst, CvPoint offset,
                              int bordertype, CvScalar value
CV_DEFAULT(cvScalarAll(0)));、
```

Offset参数告诉函数将原图像的副本放到目标图像中的位置。典型的情况是，如果核N*N是奇数时，那么边界在每一侧的宽度应是(N-1)/2,故这幅图像比原图像高N-1，这种情况下，可以吧offset设置为cvpoint((N-1)/2,(N-1)/2)。使得在每一边都是偶数项。

当使用openCV的卷积函数时，函数会自动调用上述方法，在大多数情况下，边界类型为IPL_BORDER_REPLICATE,但是有些时候我们不希望使用到他，可能会用到调用cvCopyMakeBorder方法。

- 梯度和Sobel导数

一个最重要的并且最基本的卷积是导数的计算，有很多方法可以做到，但是只有少数的方法适用于指定的方法。通常来说，用来表达微分的最常用操作是sobel微分算子。cvSobel方法定义：

```
/** @brief Calculates an image derivative using generalized Sobel
   (aperture_size = 1,3,5,7) or Scharr (aperture_size = -1) operator.
   Scharr can be used only for the first dx or dy derivative
@see cv::Sobel
*/

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

src和dst分别是输入图像和输出图像，xorder和yorder分别代表了求导阶数，通常可能是0，1，2.最多是2阶求导size的定义是滤波器宽度或者高。sobel有一个好处是可以定义任

意大小的核，并且这些核可以用快速迭代的方式构造。大核对导数有更好的逼真，小核对噪声更敏感。

注意的一点是，sobel导数并不是真正意义上的导数，因为sobel是定义到离散空间之上。也就是说x方向上的二阶sobel导数并不是真正意义上的二阶导数。他是对抛物线上的函数局部拟合。

- Scharr滤波器

事实上，在离散网络的场合下，有很多种方式可以近似的计算出导数。对于小一点的核而言，这种使用sobel算子近似计算的导数精度比较低。对于大核，由于在估计时使用了更多的点。所以这个问题并不是很严重。这种不精确性并不会出现在cvSobel使用的X和Y滤波器中表现出来。因为他们完全是按照X轴和Y轴排列。

对于一个3*3的sobel滤波器，当他的梯度角度越接近水平或者垂直方向时，这样不准确就更加明显了。OpenCV通过cvSobel函数中的CV_SCHARR的隐形使用，可以解决sobel不准确的问题。scharr滤波器和sobel滤波器一样快速，但是准确性更高，所以在使用图像度量的时候，应当首先选用scharr滤波器。滤波器原型如下所示：

-3	0	3
-10	0	10
-3	0	3

-3	-10	-3
0	0	0
3	10	3

- 拉普拉斯变换

OpenCV的拉普拉斯函数实现了拉普拉斯算子的离散模拟。

$$\triangle u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

因为拉普拉斯算子可以用二次导数定义生成，可假设其离散实现类似于二阶sobel导数，事实上，openCV在计算拉普拉斯算子的时候直接使用了sobel算子。

```
/** @brief Calculates the image Laplacian: (d2/dx + d2/dy)I
@see cv::Laplacian
*/
CVAPI(void) cvLaplace( const CvArr* src, CvArr* dst,
                       int aperture_size CV_DEFAULT(3) );
```

cvLaplace函数通常吧原图像和目标图像以及中孔大小作为变量。原图像可以是8位图像，也可以是32位图像。这里的中孔与sobel导数中出现的中孔完全一致，事实上只要给出区域大小，在二次求导的计算中，采样这个区域的像素。拉普拉斯算子可以用于边缘检测。

- Canny算子

边缘检测方法1986年由J.Canny得到了完善，也就是通常所说的Canny边缘检测法。Canny算子相比于拉普拉斯算子最大的区别在于，在Canny算子中，首先对X和Y方向上求一介求导，然后组合为四个方向的导数。这些方向导数达到； 局部最大值的点就是组成的边缘候选点。

然而，Canny算法最重要的一个特点是试图将独立的候选像素点，拼装成为图片轮廓。轮廓的形成是对这些像素运用滞后性阈值，上限和下限。如果一个像素的点，他的梯度大于上限阈值，则被认为是边缘像素，如果低于下限阈值，则被抛弃，如果在二者之间，只有当预期高于上限阈值的像素连接时才会被接收，否则抛弃。

Canny推荐的上下阈值比是2：1到3：1之间。

```
/********************************************************************************
**************\
*                                  Feature
detection                                        *
\********************************************************************************
**************/
/** @brief Runs canny edge detector
@see cv::Canny
*/

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

- 霍夫变换

霍夫变换是在图像中寻找圆、直线和简单形状的一种方法，原始的霍夫变换是直线变换，即在二值图片中寻找的相对快速的方法。变换后可以推广到普通其它情况，而不仅仅是直线。

- 霍夫线变换

霍夫直线变换的基本原理是二值化图像中的任意点都可能是候选直线集合的一部分。如果必须要对直线进行参数化，必须要找到一个斜率a和截距b， 原始图像中的点转换为平面上的（a,b）的轨迹。轨迹上的点对应着所有过原始图像上点的直线。如果我们将输入图像中所有非0的像素转换成输出图像这些点并且将其贡献相加，然后输入图像出现的直线将会在输出图像以局部最大值出现。因为我们将每个点的贡献相加，因此（a,b）平面通常被称作是累加平面。

直线方程的方程式如下：

$$r=x\cos\theta+y\sin\theta$$

openCV的霍夫变换算法并没有将这个算法显式的展现给用户。而是简单的返回了平面局部的最大值。然而，需要了解这一过程以便于更深刻的理解OpenCV中霍夫变换参数的意义。

openCV提供了不同形式的霍夫变换，标准霍夫变换（SHT）和累计概率变换（PPHT）。上文所述的是SHT算法。PPHT是这种算法的一种变种，计算单独线段的方法以及范围。之所以成为PPHT为概率算法，是因为并不将累加器平面的所有可能点进行累加，而是累加其中一部分。该想法是如果峰值足够高，只用一小部分时间去寻找他就足够了。这个猜想可以减少计算时间，openCV可以用以下同一个方法来访问这两个算法。

```
/** Variants of a Hough transform */
enum
{
    CV_HOUGH_STANDARD =0,
    CV_HOUGH_PROBABILISTIC =1,
    CV_HOUGH_MULTI_SCALE =2,
    CV_HOUGH_GRADIENT =3
};
/** @brief Finds lines on binary image using one of several methods.
   line_storage is either memory storage or 1 x _max number of lines_
CvMat, its
   number of columns is changed by the function.
   method is one of CV_HOUGH_*;
   rho, theta and threshold are used for each of those methods;
   param1 ~ line length, param2 ~ line gap - for probabilistic,
   param1 ~ srn, param2 ~ stn - for multi-scale
@see cv::HoughLines
*/

CVAPI(CvSeq*)  cvHoughLines2( CvArr* image, void* line_storage, int method,
                              double rho, double theta, int threshold,
                              double param1 CV_DEFAULT(0), double param2
CV_DEFAULT(0),
                              double min_theta CV_DEFAULT(0), double
max_theta CV_DEFAULT(CV_PI));
```

第一个参数是输入图像，必须是8位图，第二个参数是保存结果的指针，既可以是内存块，也可以是N*1的矩阵数列。下一个参数method可以是CV_HOUGH_STANDARD或者其他的。下面两个参数，rho和theta是用来设置直线的分辨率，Rho以像素为单位，rho为弧度组成的二维直方图。因此，如果平面累加可以看成是rho像素和theta弧度组成的二维直方

图，threshold是阈值，最后一个参数表示返回点的数量。

如果line_storage是访问内存的一块地址，则可以通过方法cvGetSeqElem(line,i)来获取某条直线或者线段。

- 霍夫圆变换

霍夫圆变换和之前的霍夫直线识别是类似的。说大体上是类似的，在三维中，加入了半径一个维度。这也意味着，需要耗费大量的内存和运算速度。openCV用了一种比较灵活的方式来解决这个问题。

霍夫梯度算法原理如下。首先是对图片做边缘检测，这里用的是cvCanny，然后，对边缘图像中的每一个非0点，考虑其局部梯度通过CVSobel做一阶求导，利用得到的梯度，由斜率指定的直线的每一个点都在累加器中累加，这里斜率是从一个指定的最小值到最大值的距离。然后从二维累加器中这些点选择候选中心，候选中心都大于给定阈值并且大于其相邻，这些点的累加器按照降序排列，以便于支持像素的中心首先出现。接下来对每个中心，考虑所有非零的像素，这些像素按照其与中心的距离排序。从到最大半径的最小距离高度算起，选择非零像素最支持的一条半径。如果一个中心收到边缘图像非零像素最充分的支持，并且到前期被选择的中心有足够的距离，他将会被保留。

该算法存在一些缺陷，首先， sobel算法爱可以看做是一条局部切线，并不是一个稳定的做法，有可能伴随着噪声。

其次，在边缘图像中的整个非零像素集合被认为是每个中心的候选，因此，如果累加器的阈值设置较低的情况下，算法效率会很低。并且，每个中心只能选择一个圆，这会导致同心圆无法被识别到。

最后，因为中心是被认为是按照与其关联的累加器的值升序排列的，并且太接近中心的近似圆和其它都会被抛弃，最后只会保留一个最大圆。

```
/** @brief Finds circles in the image
@see cv::HoughCircles
*/
CVAPI(CvSeq*) cvHoughCircles( CvArr* image, void* circle_storage,
                              int method, double dp, double min_dist,
                              double param1 CV_DEFAULT(100),
                              double param2 CV_DEFAULT(100),
                              int min_radius CV_DEFAULT(0),
                              int max_radius CV_DEFAULT(0));
```

霍夫圆变换和霍夫直线变换有相似的变量。输入image也是8位图，CVHoughCircles和CVHoughLine2最大的区别在于， 后面的方法需要二值化。cvHoughCircles方法会自动调用CVSobel方法，所以可以提供比较普通的灰度图像。circle_storage既可以是数组，也可以是内存指向，这取决与我们需要返回的结果。如果使用数组，则应该是CV 32FC3类型的单列数组，三个通道分别是圆的位置和半径。如果使用的是内存存储器，圆将会变成OpenCV的一个序列，CVSeq，由CVHoughCircle返回一个指向这个序列的指针。这个方法中参数必须要设置为CV_HOUGH_GRADIENT.

参数dp代表了分辨率，这个参数允许创建一个比输入图像分辨率低的累加器。如果dp设置为1，则使用原图，如果设置为2，则累加器的分辨率会影响变小，dp的值不能小于1。

参数min_dist是区分两个不同圆之间最小的边距，当使用内部canny时，param1和param2变成了cvCanny的参数。最后一个是圆的半径范围，最大和最小。

```
#include <iostream>
#include "opencv/highgui.h"
#include "opencv/cv.h"
#include "math.h"
using namespace std;
using namespace cv;
#define IMAGE_LOAD_PATH  "/Users/genesis/Pictures/circle.jpeg"
#define IMAGE_LOAD_PATH2    "/Users/genesis/Pictures/Image2.jpeg"
int main(int argc, const char * argv[]) {
    IplImage *input = cvLoadImage(IMAGE_LOAD_PATH);
    //fixme houghCircle
    IplImage *image = cvCreateImage(cvSize(input->width, input->height),
IPL_DEPTH_8U, 1);
    cvCvtColor(input, image, CV_RGB2GRAY);

    CvMemStorage* storage = cvCreateMemStorage(0);
    CvSeq * result =cvHoughCircles(image, storage, CV_HOUGH_GRADIENT, 1,
input->width / 10);

    for (int i = 0; i<result->total; i++) {
        float* p = ( float* )cvGetSeqElem( result, i );
        CvPoint pt = CvPoint(cvRound(p[0]),cvRound(p[1]));
        cvCircle(input, pt, cvRound(p[2]), CV_RGB(0xff, 0x00, 0x00));
    }
    cvShowImage("houghCircle",input);
    cvReleaseImage(&input);
    cvWaitKey();
    cvReleaseImage(&input);
    cvReleaseImage(&input2);
    cvWaitKey();
    return 0;
}
```

得到的结果：

1981年Ballard将霍夫变换拓展到任何形状，有兴趣可以了解一下。

- 重映射

许多变换都有一个共同点，具体来说，他们会把一副图像中的一个位置的像素映射到另一个位置。在这种情况下，就始终需要一些平滑的映射，但是并不能做到像素的一一对应。

我们有时候需要以编程的方式来完成这类插值，即想用一些已知的算法来确定映射。然而在一些情况下，我们想自定义映射。在深入计算映射的方法之前，我们首先看一下其他方法所依赖的可解决映射的函数。在OpenCV程序中，这样的函数是cvRemap()。

```
/** @brief Performs generic geometric transformation using the specified
coordinate maps
@see cv::remap
*/
CVAPI(void)  cvRemap( const CvArr* src, CvArr* dst,
                      const CvArr* mapx, const CvArr* mapy,
                      int flags
CV_DEFAULT(CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS),
                      CvScalar fillval CV_DEFAULT(cvScalarAll(0)) );
```

cvRemap前面的参数代表了输入图片和输出图片。显然他们的大小和通道数必须相同，但可以是任意的数据类型。重要的是要知道他们不一定是同一副图片。两个参数mapx和mapy是指具体像素重新分配的位置。他们应该与原图像的目标图像大小相同，但为单通道并且通道通常为浮点数类型（CV_DEPTH_32F）。非整型映射确定后，cvRemap将会自动计算插值。cvRemap的一个通用的用途是矫正标定和立体的图像。这些函数计算摄像头

失真并且排列为mapx和mapy参数，最后一个参数包括一个标志位flag，明确告诉了cvRemap插值如何进行。具体方式包括了：

```
/** Sub-pixel interpolation methods */
enum
{
    CV_INTER_NN        =0,        //最近邻
    CV_INTER_LINEAR    =1,        //双线性
    CV_INTER_CUBIC     =2,        //双三次插值
    CV_INTER_AREA      =3,        //像素冲采样
    CV_INTER_LANCZOS4  =4
};
```

- 拉伸收缩扭曲和旋转

本小节介绍图像的几何操作，这些操作包括了各种方式的拉伸，包括一致性缩放和非一致性缩放。 对于平面区域，有两种方式的几何变换，一种是基于2*3矩阵进行的变换，也叫做仿射变换，另一种是基于3*3矩阵的变换，又称为透视变换或者单应性映射。可以吧后一种变换当做一个三维平面被一个特定的观察者感知的计算方式，而该观察者也许不是垂直观测该平面。

一个任意的仿射变换可以表达为乘以一个矩阵在加上一个向量的形式。在OpenCV里面代表这种变换的标准形式是2*3矩阵，定义如下所示：

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \quad B = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \quad \mathrm{T} = \{\mathrm{A\ B}\} \quad X = \begin{Bmatrix} A \\ B \end{Bmatrix} \quad X' = \begin{Bmatrix} x \\ y \\ 1 \end{Bmatrix}$$

仿射变换A*X+B在效果上就等于将向量X拓展成为X’,并且只是将X’左乘以T。

仿射变换可以形象的表示以下几个形式，一个平面内的任何平行四边形ABCD可以被仿射变换为另一个平面A’B’C’D’。如果这些平行四边形的面积不等于0，这个隐含的仿射变换就被这两个平行四边形唯一定义。如果抽象的理解，就是把一幅图片放在一个胶板上，在胶板的四个角上拉或者拖，从而得到不同的四边形。

仿射变换也存在一些瑕疵，通常仿射变换是单位性建模，这是因为参数少，所以会产生一些视角微小的变化导致失真。如果不考虑这些点，仿射变换基本上满足大部分的图形集合形态变换。

透视性变换比仿射变换更加灵活，可以改变几何形状，仿射变换可以看做透视的子集。

- 仿射变换

有两种情况下会使用仿射变换。第一种是有一副想要转换的图像。第二种是我们有一个点序列并想以此计算出变化。

- 稠密仿射变换

在第一种情况下，很明显输入和输出的格式是图像。并且隐含的要求是扭曲假设对于所使用的图像，其像素必须是共稠密的表现方式。这意味着图像扭曲必须进行一些插值运算亦是输出图像平滑变得更自然一些。OpenCV提供转换函数cvWrapAffine方法。

```
/** ... and other image warping flags */
enum
{
    CV_WARP_FILL_OUTLIERS =8,   //失去的值有Fillval的方式补全
    CV_WARP_INVERSE_MAP  =16    //这个标记位用于src到dst的逆向变换
};

/** @brief Warps image with affine transform
@note ::cvGetQuadrangleSubPix is similar to ::cvWarpAffine, but the
outliers are extrapolated using
replication border mode.
@see cv::warpAffine
*/
CVAPI(void)  cvWarpAffine( const CvArr* src, CvArr* dst, const CvMat*
map_matrix,
                           int flags
CV_DEFAULT(CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS),
                           CvScalar fillval CV_DEFAULT(cvScalarAll(0)) );
```

必要条件是需要知道cvWrapAffine的性能开销问题。另一种方式是利用cvGetGuadrangieSubPix方法。这个方法可以选择的参数较少，但是优点也多。尤其是性能开销问题得到了改善，同时也可以处理多通道的函数方法。

```
/** @brief Retrieves quadrangle from the input array.
    matrixarr = ( a11  a12 | b1 )   dst(x,y) <- src(A[x y]' + b)
                ( a21  a22 | b2 )   (bilinear interpolation is used to
retrieve pixels
                                     with fractional coordinates)
@see cvWarpAffine
*/

CVAPI(void)  cvGetQuadrangleSubPix( const CvArr* src, CvArr* dst,
                                    const CvMat* map_matrix );
```

该函数是计算从src图像的点映射到dst图像上的所有点，这个映射是通过仿射变换即乘以一个2x3矩阵实现的。满足以下公式条件：

$$\mathrm{dst}(x,y) = src(a_{00}x'' + a_{01}y'' + b_{00}a_{10}x'' + a_{11}y'' + b_1)$$

观察到从（x,y）到（x”,y”）的映射产生的效果是，即使映射M是恒等映射，目标图像中心点会从原点的原图像获得。如果cvGetQuadrangleSubPix需要图像外的点，他会利用复制来重构这些点。

- 仿射映射矩阵的计算

OpenCV提供了两个函数以帮助生成映射矩阵map_matrix。如果已有两幅图像要通过仿射变换的方式发生关联或者希望以哪种方式来逼近则可以用以下函数：

```
/** @brief Computes affine transform matrix for mapping src[i] to dst[i]
(i=0,1,2)
@see cv::getAffineTransform
*/
CVAPI(CvMat*) cvGetAffineTransform( const CvPoint2D32f * src,
                                    const CvPoint2D32f * dst,
                                    CvMat * map_matrix );
```

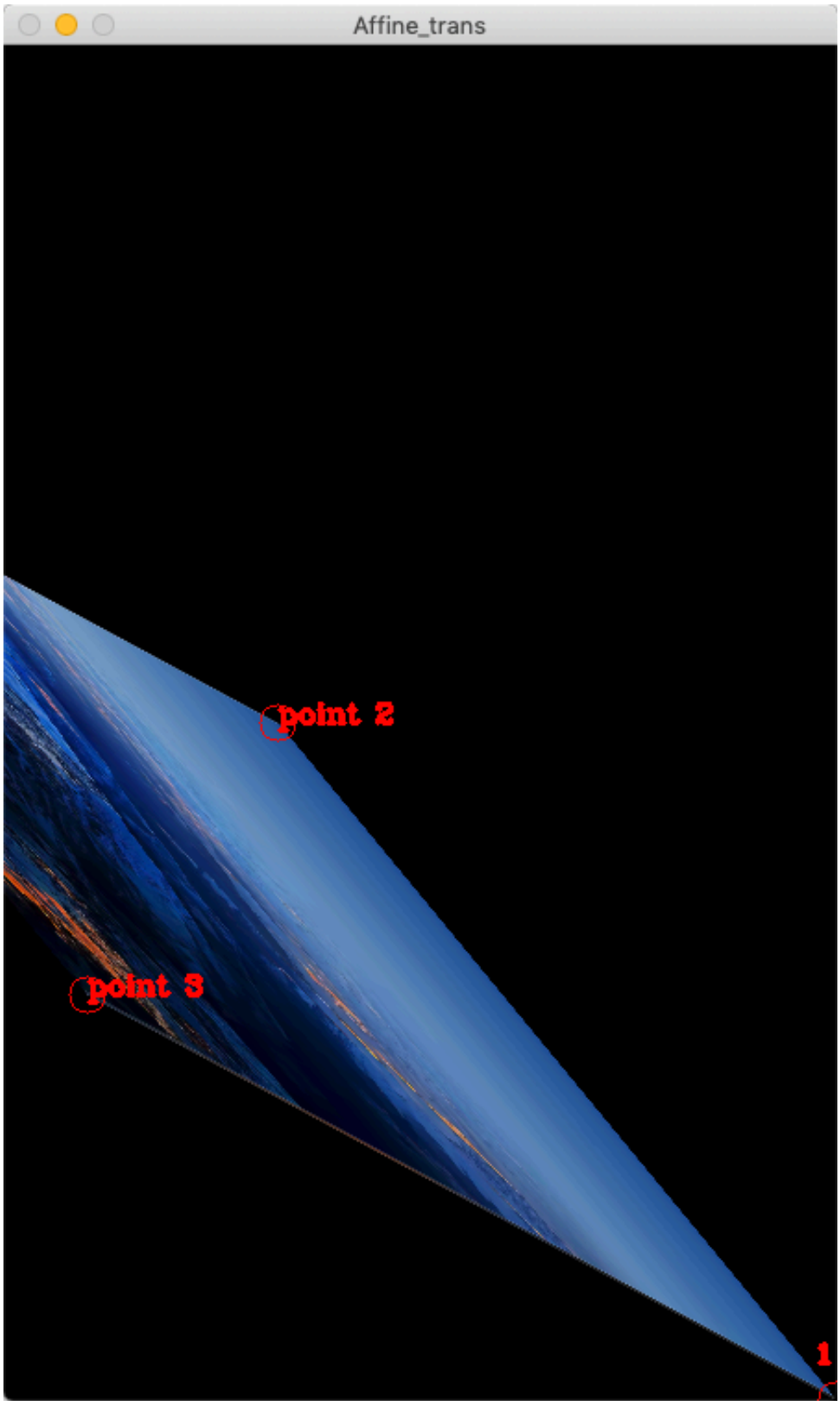
这里的src和dst是包含三个二维点（x,y）的数组，map_matrix是通过仿射变换来计算这些点。

示例：

```
#include <iostream>
#include "opencv/highgui.h"
#include "opencv/cv.h"
#include "math.h"
using namespace std;
using namespace cv;
int main(int argc, const char * argv[]) {
    IplImage *input = cvLoadImage(IMAGE_LOAD_PATH);
    IplImage *input2 = cvLoadImage(IMAGE_LOAD_PATH2);

    CvPoint2D32f srcTri[3],dstTri[3];
    CvMat* rot_mat= cvCreateMat(2,3, CV_32FC1);
    CvMat* warp_mat= cvCreateMat(2,3, CV_32FC1);

    IplImage * dst = cvCloneImage(input);
    dst->origin = input->origin;
    cvSetZero(dst);
    srcTri[0]= cvPoint2D32f(0, 0);
    srcTri[1]= cvPoint2D32f(input->width-1, 0);
    srcTri[2]= cvPoint2D32f(0, input->height-1);

    int x1 = input ->width -1;
    int x2 = input ->width *0.33;
    int x3 = input ->width * 0.1f;
    int y1 = input ->height -1;
    int y2 = input ->height *0.5;
    int y3 = input ->height * 0.7f;


    dstTri[0]= cvPoint2D32f(x1, y1);
    dstTri[1]= cvPoint2D32f(x2, y2);
    dstTri[2]= cvPoint2D32f(x3, y3);

    cvGetAffineTransform(srcTri, dstTri, warp_mat);
    cvWarpAffine(input, dst, warp_mat);
    CvFont font;
```

```
    cvInitFont(&font, CV_FONT_HERSHEY_COMPLEX, 0.5, 0.5, 1, 2, 8);
    CvPoint pt = CvPoint(x1,y1);
    cvPutText(dst, " 1", cvPoint(x1-20, y1-20),&font, CV_RGB(0xff, 0x00,
0x00));
    cvCircle(dst, pt,10, CV_RGB(0xff, 0x00, 0x00));

    pt = CvPoint(x2,y2);
    cvPutText(dst, "point 2", cvPoint(x2, y2),&font, CV_RGB(0xff, 0x00,
0x00));
    cvCircle(dst, pt,10, CV_RGB(0xff, 0x00, 0x00));

    pt = CvPoint(x3,y3);
    cvPutText(dst, "point 3", cvPoint(x3, y3), &font, CV_RGB(0xff, 0x00,
0x00));
    cvCircle(dst, pt,10, CV_RGB(0xff, 0x00, 0x00));

    cvShowImage("Affine_trans", dst);
    cvWaitKey();
         cvReleaseImage(&input);
    cvReleaseImage(&input2);
    cvWaitKey();
    return 0;
}
```

得到的实际图：



计算map_matrix的第二种方法是用cv2DRotationMatrix,他是用来计算继续围绕任意点的旋转的映射矩阵和一个可选择的尺度。虽然这只是仿射变换可能出现的一种，但是代表了一个重要的子集，有一个我们更容易理解的方式表示：

```
/** @brief Computes rotation_matrix matrix
@see cv::getRotationMatrix2D
*/

CVAPI(CvMat*)  cv2DRotationMatrix( CvPoint2D32f center, double angle,
                                   double scale, CvMat* map_matrix );
```

第一个参数是中心轴，第二个和第三个参数给出了旋转的角度和缩放的尺度。最后一个map_matrix是输出隐射矩阵，总是一个浮点类型2*3的矩阵，函数会按照如下方式计算map_matrix矩阵：

$$\begin{bmatrix} \alpha & \beta & (1-\alpha)*\text{center}_x - \beta*center_y \\ \beta & \alpha & \beta\text{center}_x + (1-\alpha)\text{center}_y \end{bmatrix}$$

可以结合使用这些设置map_matrix的方法获得图像的旋转、缩放、或者变形的效果。

- 稀疏仿射变换

如前所述，CVWarpAffine是解决密集映射的正确方法。对于稀疏映射（例如：对一系列独立点的映射），最好的方式是用cvTransform()。

一般情况下，src是D（8）通道，N*1的数组，N是将要转换的点的数量。D（8）是这些原图像点的维数。输出的目标数组dst必须相同大小但可以有不同的通道数D（d）.Nx1的数组，N是将要转换的点的数量，D（8）是这些原图像的维数，输出的目标数组dst必须要满足dst必须相同大小但是可以有不同的通道数D(d)。

```
/** Transforms each element of source array and stores
   resultant vectors in destination array */

CVAPI(void)  cvTransform( const CvArr* src, CvArr* dst,
                          const CvMat* transmat,
                          const CvMat* shiftvec CV_DEFAULT(NULL));
```

转换矩阵transmat是一个D(x)*D(d)的矩阵，被用到原图像的每一个元素，结果被植入到dst中。在前面的仿射变换中，cvTransfrom有两种用法，如何选择取决于我们想如何表达我们的转换。第一种方法中，我们变换分解成2x2部分以及2x1部分。

- 透视变换

为了获得透视变换所提供的灵活性，我们需要一个新的函数，使之能实现广泛意义上的变换。首先我们注意到，即使一个透视投影由一个单独矩阵完全确定，但这个投影实际上并不是线性变换。这是因为变换需要与最后一维元素相除，因此在处理或者运算会减少其中一唯。

与仿射变换一样，图像操作是由不同的函数来处理的相较于点集变换所用的函数，而不是通过对点集的判断。

- 密集透视变换

密集透视变换用到的OpenCV函数与提供的密集仿射变换是类似的。特别的cvWarpPerspective的参数与cvWarpAffine相同，但是有一个小的但是很重要的区别是采用了映射矩阵必须是3*3的矩阵。

```
/** @brief Warps image with perspective (projective) transform
@see cv::warpPerspective
*/
CVAPI(void)  cvWarpPerspective( const CvArr* src, CvArr* dst, const CvMat*
map_matrix,
                                int flags
CV_DEFAULT(CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS),
                                CvScalar fillval CV_DEFAULT(cvScalarAll(0))
);
```

对于一个仿射变换，在前面的代码里面，我们已经说过了填充矩阵map_matrix。有一个比较方便的函数可以计算对应的变换矩阵。

```
/** @brief Computes perspective transform matrix for mapping src[i] to
dst[i] (i=0,1,2,3)
@see cv::getPerspectiveTransform
*/
CVAPI(CvMat*) cvGetPerspectiveTransform( const CvPoint2D32f* src,
                                         const CvPoint2D32f* dst,
                                         CvMat* map_matrix );
```

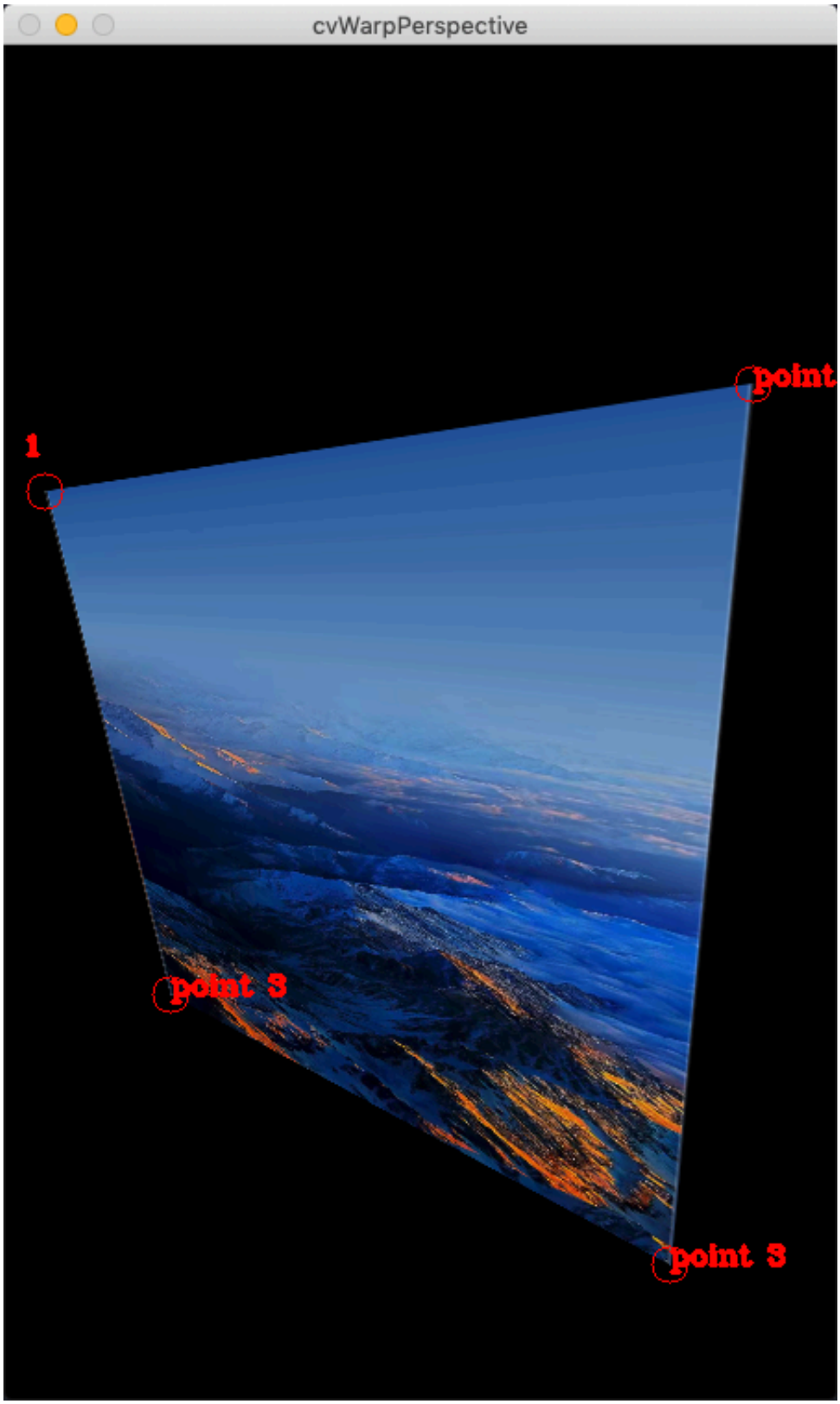
这里的src和dst是四个点的数组，所以我们能独立的控制如何将src中的矩形四角映射到dst的普通棱角上面去。我们的变换主要由原图像四个点锁指定的目标定义。如前所述的点，对透视变换，我们必须为map_matrix分配一个2x3的数组，除了3x3矩阵和三个控点要变成4个以外，其它和仿射的形式类似。

示例代码如下：

```
#include <iostream>
#include "opencv/highgui.h"
#include "opencv/cv.h"
#include "math.h"
using namespace std;
using namespace cv;
#define IMAGE_LOAD_PATH  "/Users/genesis/Pictures/background2.png"
#define IMAGE_LOAD_PATH2    "/Users/genesis/Pictures/Image2.jpeg"
#define IMAGE_SAVE_PATH  "/Users/genesis/Pictures/bmp_background_blue.jpg"
int main(int argc, const char * argv[]) {
    IplImage *input = cvLoadImage(IMAGE_LOAD_PATH);
    IplImage *input2 = cvLoadImage(IMAGE_LOAD_PATH2);

    CvPoint2D32f srcTri[4],dstTri[4];
    CvMat* rot_mat= cvCreateMat(3,3, CV_32FC1);
    CvMat* warp_mat= cvCreateMat(3,3, CV_32FC1);
```

```
    IplImage * dst = cvCloneImage(input);
    dst->origin = input->origin;
    cvSetZero(dst);
    srcTri[0]= cvPoint2D32f(0, 0);
    srcTri[1]= cvPoint2D32f(input->width-1, 0);
    srcTri[2]= cvPoint2D32f(0, input->height-1);
    srcTri[3]= cvPoint2D32f(input->width-1, input->height-1);

    int x1 = input ->width *0.05;
    int x2 = input ->width *0.9;
    int x3 = input ->width * 0.2f;
    int x4 = input ->width *0.8;
    int y1 = input ->height *0.33;
    int y2 = input ->height *0.25;
    int y3 = input ->height * 0.7f;
    int y4 = input ->height * 0.9f;

    dstTri[0]= cvPoint2D32f(x1, y1);
    dstTri[1]= cvPoint2D32f(x2, y2);
    dstTri[2]= cvPoint2D32f(x3, y3);
    dstTri[3]= cvPoint2D32f(x4, y4);
    cvGetPerspectiveTransform(srcTri, dstTri, warp_mat);
    cvWarpPerspective(input, dst, warp_mat);

    CvFont font;
    cvInitFont(&font, CV_FONT_HERSHEY_COMPLEX, 0.5, 0.5, 1, 2, 8);
    cvPutText(dst, " 1", cvPoint(x1-20, y1-20),&font, CV_RGB(0xff, 0x00,
0x00));
    cvCircle(dst, cvPoint(x1, y1),10, CV_RGB(0xff, 0x00, 0x00));
    cvPutText(dst, "point 2", cvPoint(x2, y2),&font, CV_RGB(0xff, 0x00,
0x00));
    cvCircle(dst,  cvPoint(x2, y2),10, CV_RGB(0xff, 0x00, 0x00));
    cvPutText(dst, "point 3", cvPoint(x3, y3), &font, CV_RGB(0xff, 0x00,
0x00));
    cvCircle(dst,  cvPoint(x3, y3),10, CV_RGB(0xff, 0x00, 0x00));
    cvPutText(dst, "point 3", cvPoint(x4, y4), &font, CV_RGB(0xff, 0x00,
0x00));
    cvCircle(dst,  cvPoint(x4, y4),10, CV_RGB(0xff, 0x00, 0x00));

    cvShowImage("cvWarpPerspective", dst);
    cvWaitKey();

    //fixme 清空Input Image
    cvReleaseImage(&input);
    cvReleaseImage(&input2);
    cvWaitKey();
    return 0;
}
```

示例结果：



原理算法：

```
/* Calculates coefficients of perspective transformation
```

```
* which maps (xi,yi) to (ui,vi), (i=1,2,3,4):
*
*      c00*xi + c01*yi + c02
* ui = ---------------------
*      c20*xi + c21*yi + c22
*
*      c10*xi + c11*yi + c12
* vi = ---------------------
*      c20*xi + c21*yi + c22
*
* Coefficients are calculated by solving linear system:
* / x0 y0  1  0  0  0 -x0*u0 -y0*u0 \ /c00\ /u0\
* | x1 y1  1  0  0  0 -x1*u1 -y1*u1 | |c01| |u1|
* | x2 y2  1  0  0  0 -x2*u2 -y2*u2 | |c02| |u2|
* | x3 y3  1  0  0  0 -x3*u3 -y3*u3 |.|c10|=|u3|,
* |  0  0  0 x0 y0  1 -x0*v0 -y0*v0 | |c11| |v0|
* |  0  0  0 x1 y1  1 -x1*v1 -y1*v1 | |c12| |v1|
* |  0  0  0 x2 y2  1 -x2*v2 -y2*v2 | |c20| |v2|
* \  0  0  0 x3 y3  1 -x3*v3 -y3*v3 / \c21/ \v3/
*
* where:
*   cij - matrix coefficients, c22 = 1
* input:
* 4 point
*/
double* WarpPerspective::nativeGetPerspectiveTransform(point2D* src,
point2D* dst, Matrix*&warp_mat)
{
    if (src == NULL)
    {
        LOGE("nativeGetPerspectiveTransform points error SRC");
        return NULL;
    }
    if (dst == NULL)
    {
        LOGE("nativeGetPerspectiveTransform points error dst");
        return NULL;
    }
    double* matrix = (double*) malloc(sizeof(double) * 9);
    double a[8][8] = {0};
    double b[8];
    memset(matrix, 0, sizeof(double) * 9);

#ifdef  GNDEBUG
    cout << "src dex" << endl;
    for (int i = 0; i < 4; i++)
    {
        cout << "(" << src[i].x << " , " << src[i].y << " )";
    }
    cout << "dst dex" << endl;
    for (int i   = 0; i < 4; i++)
```

```
    {
        cout << "(" << dst[i].x << " , " << dst[i].y << " )";
    }
#endif
    for (int i   = 0; i < 4; ++i)
    {
        a[i][0]     = a[i + 4][3] = src[i].x;
        a[i][1]     = a[i + 4][4] = src[i].y;
        a[i][2]     = a[i + 4][5] = 1;
        a[i][3]     = a[i][4]     = a[i][5] =
        a[i + 4][0] = a[i + 4][1] = a[i + 4][2] = 0;
        a[i][6]     = -src[i].x * dst[i].x;
        a[i][7]     = -src[i].y * dst[i].x;
        a[i + 4][6] = -src[i].x * dst[i].y;
        a[i + 4][7] = -src[i].y * dst[i].y;
        b[i]        = dst[i].x;
        b[i + 4]    = dst[i].y;
    }
    Matrix   mA  = alloc_matrix(8, 8);
    Matrix   inB = alloc_matrix(8, 1);
    for (int i   = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            set_matrix(mA, j, i, a[j][i]);
        }
    }
    for (int j   = 0; j < 8; j++)
    {
        set_matrix(inB, j, 0, b[j]);
    }
    Matrix   mX  = alloc_matrix(8, 1);
    matrixSolve(mA, inB, mX);
    for (int i = 0, n = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++, n++)
        {
            matrix[i + j * 3] = get_matrix(mX, n % 8, 0);
        }
    }
    matrix[8]  = 1.0f;
    free_matrix(mA);
    free_matrix(inB);
    free_matrix(mX);
#ifdef GNDEBUG
    print_matrix(mX);
#endif
    return matrix;
}
```

计算算子后，带入每个坐标即可得到对应的坐标：

```
int WarpPerspective::nativeWarpPerspective(argb* src, argb*&dst,
int width, int height, double* matrix)
{
    if (src == NULL || width < 0 || height < 0 || matrix == NULL)
    {
        LOGE("nativeWarpPerspective parames error ");
        return -1;
    }
    dst = (argb*) malloc(sizeof(argb) * width * height);
    memset(dst, 0, sizeof(argb) * width * height);
    for (int y = 0; y < height; ++y)
    {
        for (int x = 0; x < width; ++x)
        {
            double div   = (matrix[2] * x + matrix[5] * y +
matrix[8]);
            double src_x = (matrix[0] * x + matrix[3] * y +
matrix[6]) / div;
            double src_y = (matrix[1] * x + matrix[4] * y +
matrix[7]) / div;
            int    ux    = gnEdge(src_x, width);
            int    uy    = gnEdge(src_y, height);
            dst[ux + uy * width] = src[x + y * width];
        }
    }
    return 1;
}
```

- 稀疏矩阵变换

这里有一个特殊的函数cvPerspectiveTransform,在一系列的点上完成透视变换。我们不能使用cvTransform,因为他只局限于线性操作。因此如此，它不能处理透视变换因为透视变换需要齐次表达式的第三坐标来排除各项，需要各项注意cvPerspectiveTransform，这些数组必须是3通道浮点数类型的，矩阵mat可以是3x3也可以是4x4的。

- CartToPolar和PolarToCart

函数cvCartToPolar和cvPolarTocart方法会被一些复杂的用法用到，如cvLogPolar方法，但是他本身也是有用的。这些函数将数值在笛卡尔（x,y）空间和极性或者径向空间之间进行映射。

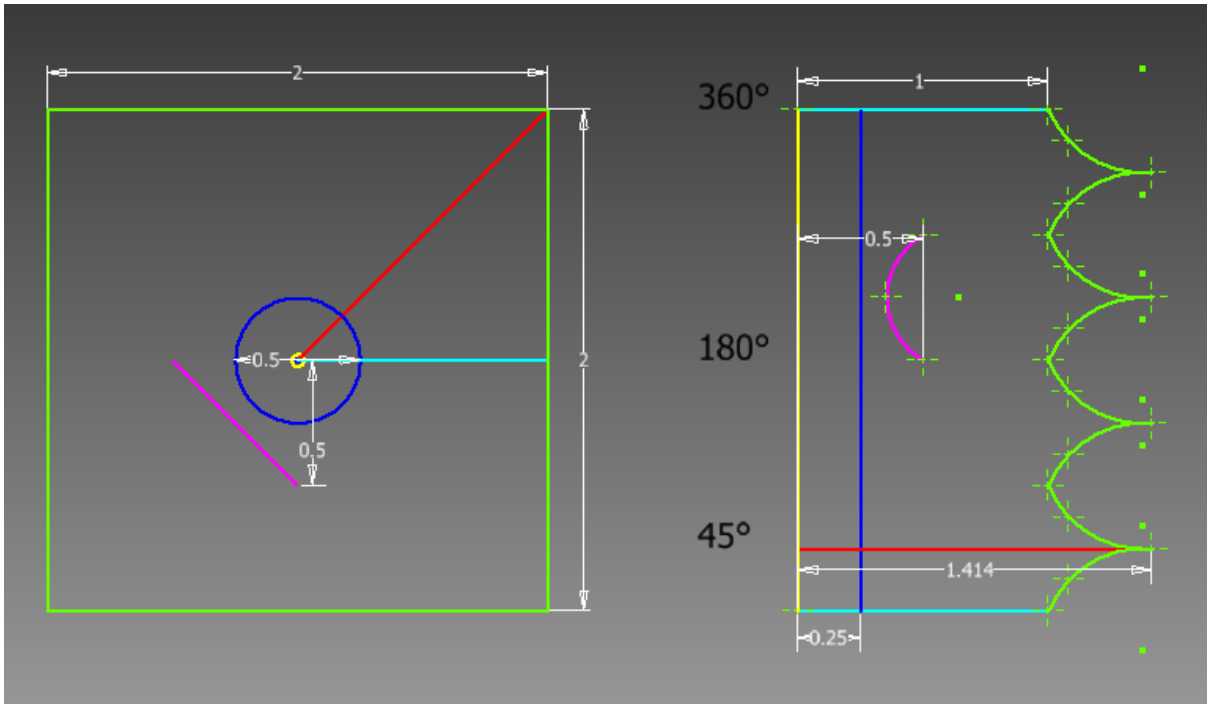
```
/** Does cartesian->polar coordinates conversion.
   Either of output components (magnitude or angle) is optional */
CVAPI(void)  cvCartToPolar( const CvArr* x, const CvArr* y,
                            CvArr* magnitude, CvArr* angle
CV_DEFAULT(NULL),
                            int angle_in_degrees CV_DEFAULT(0));
```

```
/** Does polar->cartesian coordinates conversion.
   Either of output components (magnitude or angle) is optional.
   If magnitude is missing it is assumed to be all 1's */
CVAPI(void)  cvPolarToCart( const CvArr* magnitude, const CvArr* angle,
                            CvArr* x, CvArr* y,
                            int angle_in_degrees CV_DEFAULT(0));
```

在上述两个函数方法中，前两个二维数组或者图像是输入，后两个是输出。如果输出指针设为空，他将不会被计算。这些数组必须为浮点数或者双精度切实相互匹配的。最后一个参数指定的是用角度，还是弧度计算。

- LogPolar

对于二维图像，Log-Polar转换表示从笛卡尔坐标到极坐标的变换。即（x,y）-> r e^w，其中r = √‾（x^2+y^2）并且exp(i*角度)*arctan(y/x)分离到$\alpha(\theta,\rho)$空间，而这又与中心点(x_c,y_c)有关，我们用对数运算，于是计算出$\rho = log(\sqrt{(x-x_c)^2+(y-y_c)^2})$并且计算角度值$\theta = arctan((y-y_c)/(x-x_c))$,这个方法可以模拟人类的中央视觉，并且用于物体追踪方面进行尺度及旋转的快速匹配。



对我们来说重要的是，对数极坐标变幻时对物体视场的一种不变表示，即当变换图像的质心移动到对数极坐标平面的某个固定点时。如图我们需要识别成正方形的形状，问题是看起来很不一样，有一个比其余的大很多而另一个是旋转的。对数极坐标把他变成了右边的形状，观察到平面的尺寸的差异被旋换为对数极坐标平面内沿着log(r)的位移，旋转差异被转换成为对数极坐标沿着phi轴的平移。如果我们将对数极坐标平面上每一个变换方形的转换中心，重新移动到另一个固定的中心位置，那么所有正方形都是一样的，这就产生了一类二维旋转和尺度的不变性。

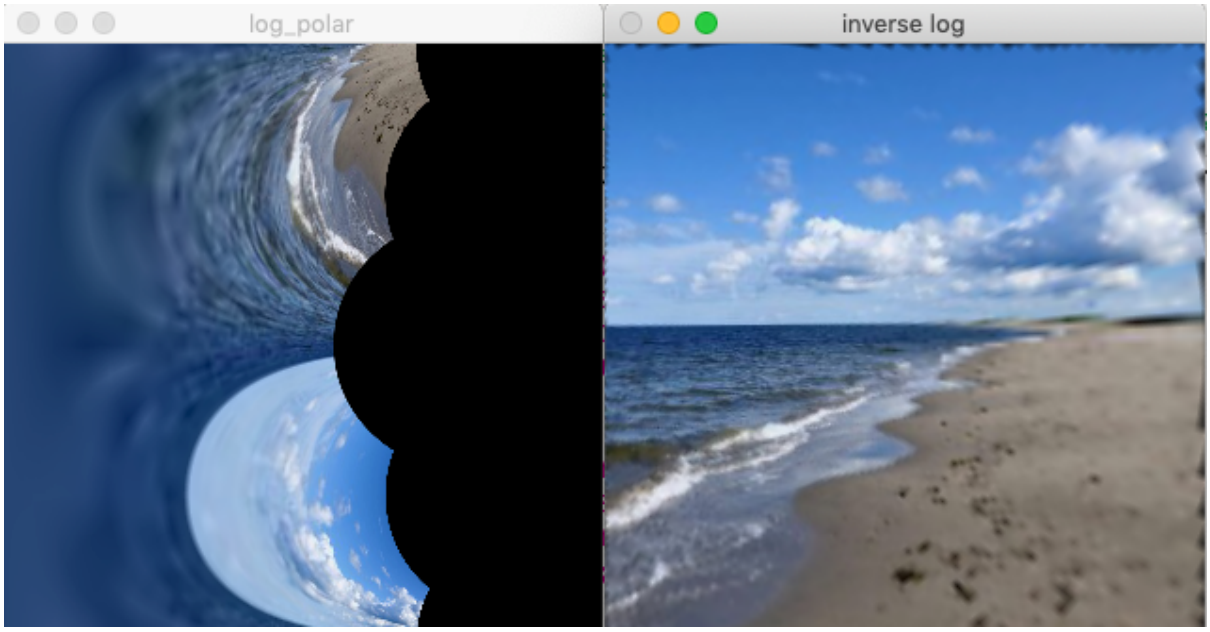
```
/** @brief Performs forward or inverse log-polar image transform
```

```
@see cv::logPolar
*/

CVAPI(void)  cvLogPolar( const CvArr* src, CvArr* dst,
                         CvPoint2D32f center, double M,
                         int flags
CV_DEFAULT(CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS));
```

src和dst是单通道或者三通道的彩色或者灰度图片。参数center是对数极坐标变换的中心点。m是缩放比例，标记位和其它稀疏矩阵算法一致。

```
int main(int argc, const char * argv[]) {
    IplImage *input = cvLoadImage(IMAGE_LOAD_PATH);
    IplImage *input2 = cvLoadImage(IMAGE_LOAD_PATH2);
    IplImage * dst = cvCloneImage(input);
    dst->origin = input->origin;
    cvSetZero(dst);
    cvLogPolar(input, dst, cvPoint2D32f(input->width/4, input->height/2),
40,CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS);
    cvLogPolar(dst, input, cvPoint2D32f(input->width/4, input->height/2),
40,CV_INTER_LINEAR+CV_WARP_INVERSE_MAP);
    cvNamedWindow("log_polar",1);
    cvShowImage("log_polar", dst);
    cvNamedWindow("inverse log",2);
    cvShowImage("inverse log", input);
    cvWaitKey();
      //fixme 清空Input Image
    cvReleaseImage(&input);
    cvReleaseImage(&input2);
    cvWaitKey();
    return 0;
}
```

所得到的结果：



- 离散型傅里叶变换DFT

对任意通过离散（整型）参数索引的数值集合，都可能用类似于连续函数的傅里叶变换的形式定义一个离散傅里叶变换DFT。对于N个复数，N(1)…N(n-1)，一维的傅里叶变换定义式子：

$$X(k)= \sum n=\{0,N-1\}x(n)\ e^{-j2\pi kn/N}$$

对于二维数组，我们可以定义以下公式：

$$F(u,v)=1/MN\sum_{x=0}^{M-1}\left[\sum_{y=0}^{N-1}f(x,y)\exp(-j2\pi vy/N)\right]\exp(-j2\pi ux/M)$$


```
void idft(double** re_array, double** im_array, short** out_array, long
height, long width)
{
    double real, temp;

    for (int i = 0; i < height; i++){
        for (int j = 0; j < width; j++){
            real = 0;

            for (int x = 0; x < height; x++){
                for (int y = 0; y < width; y++){
                    temp = (double)i * x / (double)height +
                           (double)j * y / (double)width;
                    real += re_array[x][y] * cos(2 * pi * temp) -
                            im_array[x][y] * sin(2 * pi * temp);
                }
            }
            out_array[i][j] = (short)(real / sqrt(width*height));
            if (out_array[i][j] > 0xff)
                out_array[i][j] = 0xff;
            else if (out_array[i][j] < 0)
                out_array[i][j] = 0;
        }
    }
    printf("idft done\n");
}
```

事实上，cv提供了多种快速傅里叶FFT算法能够以N log N的时间复杂度来计算这些值。

- 频谱乘法

在许多包含计算DFT的应用中，还需要得两个频谱的元素相乘。由于DFT的结果是以共特殊的高密度格式封装，并且通常是复数，解除他们的封装以及通过普通矩阵来进行乘法运算是很乏味的。openCV提供一个方便的函数，cvMulSpectrums。他可以准确的执行这个函数以及执行一些其他方便的事情。

```
/** Multiply results of DFTs: DFT(X)*DFT(Y) or DFT(X)*conj(DFT(Y))
@see core_c_DftFlags "flags"
*/

CVAPI(void)  cvMulSpectrums( const CvArr* src1, const CvArr* src2,
                             CvArr* dst, int flags );
```

前面两个是src参数，dst是输出，flags是需要函数执行什么动作的事情。

- 卷积和DFT算法

利用DFT可以大大的加快卷积运算的速度，减少在卷积运算中的性能损耗。在空间域中的卷积就是在频域中的相乘运算。为了实现这个，必须要对图形图像进行傅里叶变换，接着计算卷积滤波器的傅里叶变换。一旦完成这个，可以在变换域中以相对于原图像素数目的线性时间内进行卷积运算。值得看此类卷积运算的源代码，他也将为我们提供很多cvDft的用法案例。

- 离散型余弦变换

对于实数，通常计算离散傅里叶变换的一半已经足够了。离散余弦变换DCT与DFT类似。参考计算公式：

正变换：

$$F(u,v)=\alpha(u)\alpha(v)\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}f(x,y)\cos\frac{(2x+1)u\pi}{2M}\cos\frac{(2y+1)v\pi}{2N}$$

其中，u=0,1.....M-1,　v=0,1.....N-1.

$$\alpha(u)=\begin{cases}\frac{1}{\sqrt{M}} & u=0\\ \sqrt{\frac{2}{M}} & u\neq 0\end{cases}\qquad \alpha(v)=\begin{cases}\frac{1}{\sqrt{N}} & v=0\\ \sqrt{\frac{2}{N}} & v\neq 0\end{cases}$$

反变换：

$$f(x,y)=\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}\alpha(u)\alpha(v)F(u,v)\cos\frac{(2x+1)u\pi}{2M}\cos\frac{(2y+1)v\pi}{2N}$$

其中，x=0,1.....M-1,　y=0,1.....N-1.

$$\alpha(u)=\begin{cases}\frac{1}{\sqrt{M}} & u=0\\ \sqrt{\frac{2}{M}} & u\neq 0\end{cases}\qquad \alpha(v)=\begin{cases}\frac{1}{\sqrt{N}} & v=0\\ \sqrt{\frac{2}{N}} & v\neq 0\end{cases}$$

Cv提供了函数方法：

```
/** Discrete Cosine Transform
```

```
@see core_c_DftFlags "flags"
*/
CVAPI(void)  cvDCT( const CvArr* src, CvArr* dst, int flags );
```

- 积分图像

OpenCV可以用轻松计算积分的图像，只要用一个具有相应名称cvIntegral的函数。积分是一个数据结构，可实现子区域的快速求和。这样的求和在许多应用中是很有用的，最显著的是在人脸识别和相关算法中的haar小波。

```
/** @brief Finds integral image: SUM(X,Y) = sum(x<X,y<Y)I(x,y)
@see cv::integral
*/
CVAPI(void) cvIntegral( const CvArr* image, CvArr* sum,
                        CvArr* sqsum CV_DEFAULT(NULL),
                        CvArr* tilted_sum CV_DEFAULT(NULL));
```

- 距离变换

图片的距离变换被定义为一副新图片，该图片的每个输出像素都被设置成为输入图像与0像素最接近的距离。显然，典型的距离变换的输入应该是某些边缘检测图像。在多数应用中，距离变换的输入时例如canny边缘检测的输出。

在实际中，距离变换通常是利用3X3和5x5的数组淹磨进行的。数组中每个点被定义为这个特殊位置同其相关的淹膜中心的距离。较大的距离以由整个淹膜定义的动作序列形式被建立。这意味着要用更大的淹膜生成更精准的距离。

cvDistTransform中可能采用的距离标准，另外，可以定义一个特殊的标准及与之相关的特殊矩阵。

cvDistTransform中可能采用的距离标准：

```
/** Distance types for Distance Transform and M-estimators */
enum
{
    CV_DIST_USER    =-1,  /**< User defined distance */
    CV_DIST_L1      =1,   /**< distance = |x1-x2| + |y1-y2| */
    CV_DIST_L2      =2,   /**< the simple euclidean distance */
    CV_DIST_C       =3,   /**< distance = max(|x1-x2|,|y1-y2|) */
    CV_DIST_L12     =4,   /**< L1-L2 metric: distance = 2(sqrt(1+x*x/2) -
1)) */
    CV_DIST_FAIR    =5,   /**< distance = c^2(|x|/c-log(1+|x|/c)), c =
1.3998 */
    CV_DIST_WELSCH  =6,   /**< distance = c^2/2(1-exp(-(x/c)^2)), c =
2.9846 */
    CV_DIST_HUBER   =7    /**< distance = |x|<c ? x^2/2 : c(|x|-c/2),
c=1.345 */
};
/** @brief Applies distance transform to binary image
@see cv::distanceTransform
```

```
*/
CVAPI(void)  cvDistTransform( const CvArr* src, CvArr* dst,
                              int distance_type CV_DEFAULT(CV_DIST_L2),
                              int mask_size CV_DEFAULT(3),
                              const float* mask CV_DEFAULT(NULL),
                              CvArr* labels CV_DEFAULT(NULL),
                              int labelType
CV_DEFAULT(CV_DIST_LABEL_CCOMP));
```

- 直方图均值化

通常情况下，摄像机和图像传感器必须不仅仅处理场景的对比度、还必须让这个场景的灯光曝光度保留到结果里。直方图均值化的原理是一个分布转到另一个分布上（一个更宽，统一的亮度值分布）。也就是说，我们希望把原始的分布中y轴的值在新的分布中尽可能的展开。在彩色图中，最好采用HSV的物理原型分布处理，灰度图可以按照原理实现。

```
static void HistgramAverragehsv(AndroidBitmapInfo* info, argb*
inpixels,
                                 argb* outpixels)
{
    int   i, j;
    float Average[256];

    int  BmpWidth  = info->width;
    int  BmpHeight = info->height;
    hsv  hsvs;
    hsv  hsvout;
    argb out;
    for (i = 0; i < 256; i++)
    {
        Average[i] = (float) display[i] / (BmpWidth * BmpHeight);
        LOGI("%f", Average[i]);
    }
    for (i = 0; i < BmpWidth * BmpHeight; i++)
    {

        RGB_TO_HSV(&inpixels[i], &hsvs);
        int temp  = (int) (hsvs.V * 255);
        int vtemp = 0;
        for (j = 0; j < temp; j++)
        {
            vtemp = (int) (255 * Average[j] + 0.5) + vtemp;
        }
        if (vtemp > 255)
        {
            vtemp = 255;
        }
        hsvout.V = (float) vtemp / 255;
        hsvout.H = hsvs.H;
```

```
        hsvout.S = hsvs.S;
        HSV_TO_RGB(&hsvout, &out);
        outpixels[i].red   = out.red;
        outpixels[i].green = out.green;
        outpixels[i].blue  = out.blue;
        if (outpixels[i].red > 255)
            outpixels[i].red   = 255;
        if (outpixels[i].blue > 255)
            outpixels[i].blue  = 255;
        if (outpixels[i].green > 255)
            outpixels[i].green = 255;
        outpixels[i].alpha = (int) inpixels[i].alpha;
    }
}
```